DOCUMENT RESUME

ED 135 327 IR 004 276

AUTHOR Cannara, A. B.
TITLE Experiments in Teaching Children Computer Programming. Technical Report No. 271. Psychology and Education Series.
INSTITUTION Stanford Univ., Calif. Inst. for Mathematical Studies in Sccial Science.
SPONS AGENCY National Science Foundation, Washington, D.C.
PUB DATE May 76
GRANT NSF-FJ-443X
NOTE 134p.; For related document see ED 111 347; Ph.D. Dissertation, Stanford University
AVAILABLE FROM Stanford University, Institute for Mathematical Studies in the Social Sciences, Stanford, California 94305

EDRS PRICE MF-$0.83 HC-$7.35 Plus Postage.
DESCRIPTORS *Computer Assisted Instruction; Elementary Education; Experimental Programs; *Experimental Teaching; *Programing
IDENTIFIERS LOGO; Simper

ABSTRACT

Two experiments are conducted to observe how children, ages 10-15, who have not previously used a computer, will learn concepts relevant to computer programing languages. The interpreters used are LOGO and SIMPER. Subjects are pretested with an instrument developed to predict ability to manage the concepts. They are then given group instruction on a weekly basis with individual practice four days a week. Teletypes, for the most part, are used to interact with the computer, and tutorial assistance is given. After the first experiment the crder of presentation is altered. The results of both experiments lead to the identification of changes that are needed in the interpreters and to suggestions for further revisions in curriculum design. The report is supported by tables, and by appendices including a third interpreter, SPM, and aptitude testing details. (WBC)

EXPERIMENTS IN TEACHING CHILDREN COMPUTER PROGRAMMING

by

A. B. Cannara

TECHNICAL REPORT NO. 271
May, 1976

PSYCHOLOGY AND EDUCATION SERIES

INSTITUTE FOR MATHEMATICAL STUDIES IN THE SOCIAL SCIENCES
STAN[illegible]RD UNIVERSITY
STANFORD, CALIFORNIA 94305

Foreword

This dissertation is actually an extension and continuation of an earlier technical report (Weyer & Cannara, 1975), which herein is called "report-1". Report-1 should (but need not) be read in conjunction with this document.

As a matter of notation, all phrases in the text that are not in English, but in the computer languages being discussed, are surrounded by single quotes ('), unless their context is otherwise obvious. As a matter of taste, which rejects the "arrogance of the acronym", names of programming languages and other proper nouns, excepting trade or institutional names, receive their due quota of capitals--one.

Several references are made to articles in a few common periodicals, not necessarily because the articles are uniquely relevant, but because the periodicals are easily found and their editors traditionally strive for clarity as well as accuracy.

Acknowledgements

This work was supported by National Science Foundation Grant NSF-GJ-443X. I thank Patrick Suppes for his helpful advice and for use of the IMSSS computing facilities. I am further indebted to my student volunteers. They remain anonymous, but they were the most important people in the experiments. Steve Weyer deserves thanks for his imaginative work on the original Logo curriculum, for maintaining the Logo interpreter and for developing the Sailogo graphics system. I also thank Adele Goldberg for her editorial assistance.

Table of Contents

Index of Tables

Index of Figures

1. Introduction

Herein are discussed in detail two experiments done at the Institute for Mathematical Studies in the Social Sciences (IMSSS) at Stanford during the summer of 1973 and the spring of 1974. Previously, Cannara and Weyer (1974a), and Weyer and Cannara (1), have described the 1973 experiment; it will be referred to as the "first" experiment. The 1975 document will be called "report-1".

The experiments attempted to study children, who had never used a computer before, learning: (a) concepts relevant to computer programming, and (b) modern programming languages. The languages and other programming facilities used (e.g., graphics) have been discussed definitively in report-1; this thesis will simply outline their features and concentrate on observations of the childrens' learning-processes and the implications of both experiments in terms of programming-language and curriculum design, and tutoring technique.

Why observe children learning computer-programming? Programming would seem to be a decidedly adult task for young people who haven't yet completed their basic schooling. A partial, motivational answer follows immediately; the remainder of this thesis may be viewed as an attempt to complete that answer.

Three main streams of thought converge on the study of children learning to use a computer. First, is the view that a computer is in fact a tool for thinking, which implies that it might be applied fruitfully at every educational level (e.g., Brown & Rubinstein, 1973; Dwyer, 1972; Feurzeig, Papert, Bloom, Grant & Solomon, 1969; Kay, 1972[b]; Papert, 1970). In particular, the computer can be used to stimulate the activity Papert has referred to as "thinking about thinking". Second, is the desire of some educators to study the thinking processes of people solving problems (e.g., Bloom & Broder, 1950; Piaget, 1970; Polya, 1957), which leads directly to studies in perhaps the most

general problem-solving realm: computer programming. Third, is a synthesis of human problem-solving and computer-as-tool, flamboyantly named "artificial-intelligence research", which aims to formalize problem-solving procedures (e.g., Feigenbaum & Feldman, 1963; Minsky, 1968; Newell & Simon, 1972; Nilson, 1971; Winograd, 1971). Sometimes, artificial-intelligence products can better our understanding of ourselves and/or provide useful educational strategies (e.g., Brown & Burton, 1974; Goldberg, 1973).

We should not be surprised to find children and computers where those three streams intersect. The educational linkage of computer programming with thinking is expressed by the idea of a "mathematical laboratory", in which a program creates a constructively interactive, and so perhaps more interesting environment for learning. Unlike most traditional realizations of computer-assisted-instruction (Cai), the laboratory is designed to "understand", at a meaningful level, the domain of interest.[1] Unlike most classroom lectures, such a laboratory can give substance to the material and exploratory freedom to its users. The user's interaction with a mathematical laboratory is mediated by a formal (as opposed to natural) language, whose semantics access the constructive abilities of the laboratory and whose syntax is simply a perhaps novel set of conventions. A computer and a programming language together constitute a mathematical laboratory of the most general kind, because they are all that are needed to construct (simulate) any other laboratory. That is the main justification for studying programming as a general problem-solving activity. It is based upon a conjecture of Church's that (freely interpreted) suggests that any ideas which may be formalized may be studied as a computer program.[2] Formalization of ideas, a fundamental aspect of mathematics, is part and parcel

---

[1] See Ellis (1974) or Dettinger and Marks (1969), for critiques of present educational computer applications, and Smallwood (1962), or Suppes (in Wittrock, 1973), for historically typical examples of CAI.

[2] For discussions of Church's thesis, see Manna (1972) or Minsky (1967).

of mathematical laboratories for programming, problem solving and thinking about thinking.

The production of effective mathematical laboratories is closely and bidirectionally connected with artificial-intelligence work and human self-understanding. It must grapple with questions beyond the immediate scope of the laboratories themselves. Effective tutoring techniques, for example, are educational objectives which must be attained even after one has constructed a laboratory which "understands" its domain. Thus, applying the computer educationally, as a tool made available via a laboratory, demands answers to questions posed in a wide variety of fields (e.g., Brown & Burton, 1974; Goldberg, 1973).

The theory and practice of computation offer educators some valuable tools: (a) the formalization of ideas as clustered sequences of instructions, (b) methods for modelling real-world processes, and (c) metaphors for understanding machine and human information-processing. Together, these expose thinking techniques that Papert has termed "powerful ideas". Concepts of programming and thinking can be taught as natural and inseparable partners, emphasizing students' scrutiny of their own thinking about the world. And, it is not a new idea that school-children can and should learn how to program a computer, so that they too might access its unparalleled power as a tool for thinking. The computer's natural ability to simulate has responded to the ingenuities of students (as seen, for example, in the work of Brown and Rubinstein, Dwyer, or Papert) with the same spectacular generality it has provided to professional researchers (e.g., Levison, Ward & Webb, 1973; Toomre & Toomre, 1973; Winograd, 1971).

The foregoing remarks were intended to justify a desire to study programming as an intellectual activity for children and programming languages

as tools for such activity. As a technological product, the personal computer will soon be as much a reality, as the personal calculator is today.[3] Access to interactive computation may soon become commonplace for vast numbers of children (and adults), at school or at home. Certainly we should be trying now to understand how to hone this new tool to maximum usefulness. As a medium for expressing, manipulating and communicating ideas, the personally accessible computer may stand well above everything since the printing press.[4]

Teaching programming is a tutorial endeavor of perhaps the most general kind. The work to be reported here attempts to characterize some of the situations that human and mechanical tutors for programming will confront and must be prepared to resolve. It is relevant to common ground between education and artificial intelligence because the production of computer programs which can tutor humans with human proficiency is a common goal. No one has attained that goal yet, because the activities of a good tutor are tied irrevocably to humanness of language and knowledge (e.g., Winograd, 1974). Although the theoretical power of the computer (i.e., as conjectured by Church) may be sufficient to simulate natural intellect, we do not yet understand ourselves (or other species, e.g., Gardner & Gardner, 1975) well enough to communicate even a coarse description of intelligence to any recipient (note the arguments of Stent, 1975a, 1975b; and his critics). Those who have recognized the nature of this problem have come closest to success in carefully limited contexts (e.g., Brown & Burton, 1974; Carbonell, 1970; Winograd, 1971).

The generality of a programming laboratory and the intimacy of tutoring combine to produce an interesting research environment in which analysis of

---

[3] See Kay (1975, 1972a, 1972b) or Brand (1974, pp. 64-71) for one view of the near future of computing.

[4] See Fenichel & Weizenbaum (1971), and Manchly (1975) for historical perspectives on the computer's development, and Zacroux (1975) for directions that technological events are taking now.

errors plays a central role. A programming tutor (human or mechanical) must be ready to intelligently suggest, accept or comment on an arbitrarily wide range of student interactions and program syntheses. The details of errors do more than indicate what a student does not understand, they indicate how the student views the problem at hand in terms of his or her own view of the world.

Extending a suggestion of Papert's, if a student responds to a posed problem at all, that response is typically correct by the student's personal analysis. So the student is surprised to hear "wrong". It is the tutor's responsibility to try to divine the reasons for the student's error, perhaps acting as does a detective eliciting evidence from someone from a foreign land--subsequent interaction is devoted to laying a common foundation of terms (definitions and relations). The tutor necessarily learns about the student's world view and is better prepared to handle future errors and future students. Errors are not "bad", they provide valuable feedback to be exploited for student benefit.[5]

However, any tutor (human or mechanical) for teaching something as general as programming is destined to occasionally fail the student, because it must occasionally tackle unsolvable (uncomputable) problems.[6] In other words, the tutor must pass judgment on the correctness of a student's program, and we know that there exists no general procedure for deciding that an arbitrary program is correct or incorrect. But the range of solvable problems is so broad that this hard theoretical fact discourages neither researchers nor teachers. "Proof of program correctness" (Hoare, 1971) and "automatic program synthesis"

---

[5] This relates to a basic criticism of most past efforts in Cai: not only have programs been designed which fail to understand their own subject-matter, they fail to possess more than trivial error-handling strategies. Results too often have been just transfer of programmed instruction text or film to computer storage, using very little, from the student's vantage, of the computer's computational potential. Dwyer has said that Cai fails in "reproducing the excitement of masterful teaching". I would add that rarely have Cai workers even attempted to capture masterful teaching.

[6] Discussions of the uncomputable (unsolvable or unprovable) appear in Davis (1965), Minsky (1967), Chaitin (1975) and Steen (1975).

(Fenichel, Weizenbaum & Yochelson, 1970) are active topics in computational research which have clear bearing on future success in constructing competent computer-based tutorial systems.

Numerous research projects have taught children particular programming languages (e.g., Feurzeig and Lukas 1972a; Fischer, 1973; Folk, Statz & Seidman, 1974; Milner, 1973; Roman, 1972). However, apparently none has attempted to make explicit the broad range of relevant programming concepts and their relationship to a student's world of thought. In such terms, many projects have pursued hazy, sometimes arbitrary goals that concentrated on teaching an available language through ad-hoc, problem-solving situations, without generalizing situations and solution strategies. A study by Folk, et al., (1974) is perhaps the most extensive attempt to specify relationships between programming concepts and children's thinking processes. But their analysis is confined to classical statistical models and the concomitant testing of rather broad hypotheses virtually ignores a wealth of detail in student protocols.

In contrast, protocols (and tutorial notes) are precisely the data upon which this work is founded. The primary objective is to understand how children learn programming concepts (e.g., Table I), with secondary emphasis on the influences of languages and curricula. With error-analysis as a tool, student/machine interactions must be exposed in as much detail as possible. Narrow views, provided for example by conventional test scores, are inadequate no matter how convenient they may be to obtain and analyze. Quoting Bloom and Broder on the subject of "objective" tests:

> *"What is missing is information on the process by which the problems are solved. The methods of attack, the steps of the thinking process, the kinds of considerations used to make one choice rather than another, and the feelings and attitudes of the subject are neglected or given very little attention.*
>
> *"... attention on the processes of thought...may also require a change from testing and mass studies to those which involve small numbers of subjects studied by rather intensive techniques.*

Table I

Some Fundamental Programming Concepts

1. Machine as a tool manipulated with a command language
2. Machine possessing an alterable memory
3. Literal expressions
4. Name-value associations
5. Evaluation and symbol-substitution
6. Execution of stored programs
7. Programs which make decisions
8. Procedures (algorithms)
9. Evaluation of arguments to procedures
10. Procedures as realizations of functions (transformations)
11. Composition of functions
12. Partial and total functions
13. Computational context (local versus global environments)
14. Evaluation in changing environments
15. Induction (recursion and iteration)
16. Data structures as defined by functions
17. Problem formulation (representation)
18. Incomplete algorithms (heuristics)

"The way in which each student looks at a particular task may make it a unique problem for him."

-- Bloom & Broder, (1950).

It may seem obvious that to understand a physical or intellectual process one must exercise and observe it. In fact one must observe what it does wrongly as well as correctly before a good model of the process' structure can be realized. Thus has error analysis proven its value in many fields (e.g., Fromkin, 1973; Newell & Simon, 1972). It is a basic means for evaluating theories in all the sciences.

"Truth arises more easily from error than from confusion."

--Francis Bacon.

This work has depended upon observing children learning by making mistakes and discoveries. For their own benefit and for the practical requirements of research, the children had to feel motivated and supported. Motivation is an essential precursor of effective learning, yet it is often snubbed in the analysis of everyday education (Jackson, 1968); and it has yet to be captured accurately in artificial-intelligence applications. So, apart from examining interactions with a programming laboratory, this work has also been concerned with the motivational aspects of tutoring, curricula, languages and concepts.

That programming concepts provide a link between formalized thinking and perceived reality is certainly not a new axiom (Berry, 1964). It was assumed, perhaps tacitly, in much of the similar research quoted earlier. For motivation, a student should look to his or her own life experience for applications of the tools which an understanding of pertinent concepts supplies. This is the ultimate justification for teaching programming, because the power of a programming laboratory derives from the fact that students do more than

interact with it, they intervene, and mold the laboratory to their very own purposes.

The research problem can be summarized by two questions: (a) How do the characteristics of a programming laboratory influence a child's motivation and ability to learn programming concepts and apply them to the solution of problems? and (b) What are some significant features of that learning process?

2 Programming Facilities

Both experiments attempted to impart an understanding of the concepts in Table I and fluency in two, very different programming languages. This required the development of: (a) interactive laboratories (interpreters) for the languages and devices used, (b) parallel curricula for teaching the concepts, (c) means for acquiring data on each student's interactions, and (d) means for judging each student's aptitude for programming and mastery of the concepts.

Part of requirement (a) was met easily by using existing interpreters for two languages, Logo and Simper, developed specifically to teach children computer programming. At one time, a third language (Spm, Appendix 1), designed by the author, was also a candidate but was discarded. Development of some of the devices used and requirements (b), (c) and (d) defined the work to be done preliminary to the experiments.

Languages

The languages Simper and Logo were chosen because they are computationally general, they are relatively easy to learn, they are interactive with powerful editing features, and they are highly dissimilar (Figure 1). Both are detailed extensively in report-1, so only a brief description is necessary here. Both experiments, the first (summer-1973) and second (spring-1974), led to changes in both languages--these will be indicated also. In the text, paired, single quotes (') denote items in the Logo and Simper languages.

Simper was developed by Lorton and Slimick (1969) at IMSSS as a simple simulation of an imaginary machine resembling an Hewlett-Packard model 2000. It has been used to teach business applications of programming to students at Woodrow Wilson High School in San Francisco (Lorton & Muscat, 1975). At IMSSS, it has been expanded and rewritten in the Algol-60 subset of Sail (Swinehart and Sproull, 1971) by the author.

Simper	Logo
001 :PUT A 43	←TO REPEAT :LETTER:
002 :NAME REPEAT	@10 TYPE :LETTER:
002 !CWRITE A	@20 REPEAT :LETTER:
003 :PUT P REPEAT	@END
004 :RUN	REPEAT DEFINED
EXECUTING 1 TO 500	←REPEAT "+"
++++++++++↑G	++++++++++↑G
...23 INSTRS IN .043 SEC.	I WAS AT LINE 10 IN REPEAT
004 :EDIT 1	←EDIT REPEAT
001 !CASK A	@EDIT TITLE
004 :SLIDE 2:7	@TITLE TO REPEAT :LETTER: :TIMES:
002 :ASK B	@5 TEST LESSP :TIMES: 1
003 :NEGATE B	@7 IFTRUE DONE
004 :JUMP B .+2	@EDIT LINE 20
005 :HALT	20 REPEAT :LETTER: DIFFERENCE :TIMES: 1
006 :INCREMENT B	@END
007 !NAME 4 REPEAT	REPEAT DEFINED
SWITCHING REPEAT'S REFERENCES	
007 !RUN	←REPEAT "+" 10
	++++++++++←EDIT REPEAT
EXECUTING 1 TO 500	@6 IFTRUE SKIP
+10	@END
++++++++++	REPEAT DEFINED
HALT...45 INSTRS IN .117 SEC.	
	←REPEAT "+" 10
007 !LIST	++++++++++
	←LIST REPEAT
YOUR PROGRAM:	
	TO REPEAT :LETTER: :TIMES:
001 :CAS A	5 TEST LESSP :TIMES: 1
002 :ASK B	6 IFTRUE SKIP
003 :NEG B	7 IFTRUE DONE
004 :JUM B .+2 (REPEAT)	10 TYPE :LETTER:
005 :HAL	20 REPEAT :LETTER: DIFFERENCE :TIMES: 1
006 :INC B	
007 :CWR A	
008 :PUT P REPEAT	

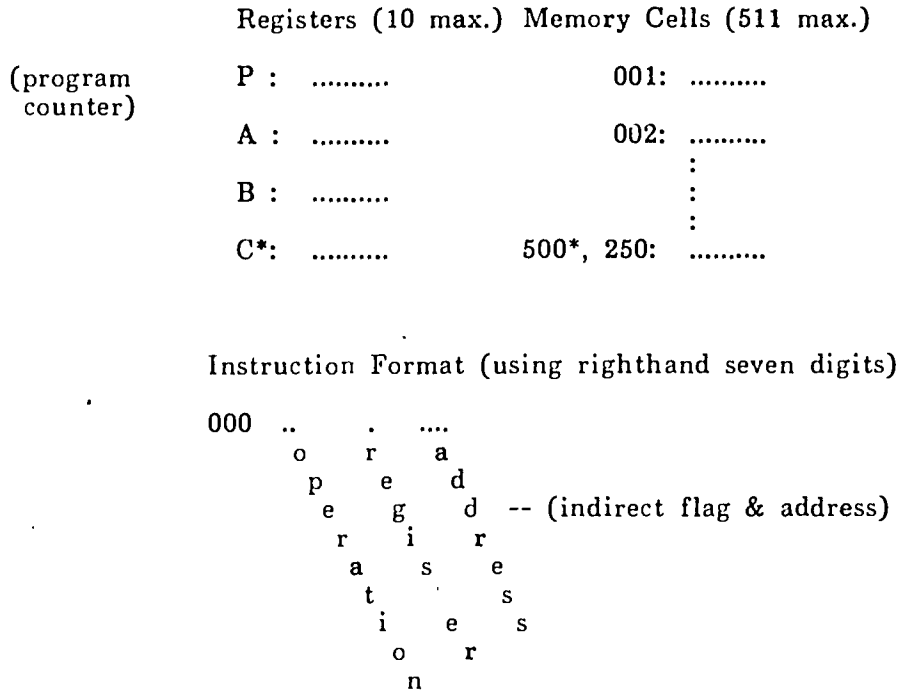
(These sample dialogues produce alternative programs for the repeated printing of a keyboard character supplied by the typist. Prompts from Simper are the current memory address (a decimal numeral) and a ":" or an "!", depending on whether the addressed location is empty or used. Logo prompts "←" at the outer level and "@" at the editing level. "↑G" indicates a control character typed to stop a potentially endless execution sequence.)

Fig. 1. Simper and Logo Sample Dialogues

Simper, is designed for interactive use. It is an assembly-language interpreter for a simple decimal machine with an addressable program counter. Its instruction set typifies those of early minicomputers and is similar to, but simpler than, that of the language Mix (Knuth, 1970). As a programming laboratory, Simper has three functional components: (1) a simulator for the underlying machine (Figure 2), (2) a real-time assembler which translates symbols and mnemonic instructions (listed in Table II) into machine language, and (3) an interpreter which handles editing and general management of programs (Table III). This system allows students to generate and easily "debug" nontrivial machine-language programs. One can imagine that, when the Simper interpreter is not running a user's program, it is simply waiting for a message from the user which is either a phrase in one of the three languages: machine, assembly or interpreter, or is unintelligible. The reader should examine Figure 1 again, and then try to follow the execution of the sample program (which realizes the function: 2x + 9) in Figure 3.

Logo (Feurzeig, et al., 1969) is a procedural language whose basic data structures are strings of letters or words. The Logo instruction-set is easily expanded via procedure (operation) definitions, possibly recursive. An important feature of Logo (as opposed to Fortran-like languages) is that operations which a user defines are syntactically equivalent to Logo primitives. Logo contains essentials of the currently popular Basic language as a subset, but is superior to Basic in terms of mathematical consistency, and clarity of phrasing and control. Furthermore, Logo begins to address the important question of language extensibility, which is a fundamental measure of the usefulness people can attribute to any language for computing or thinking.

The Logo interpreter used in these experiments was obtained from Bolt, Beranek & Newman Inc. (BBN) of Boston. It is written in Macro assembly-language for the PDP-10. For the purposes of the experiments, Logo was



(* indicates the configu ation after the first experiment. The machine simulated within the Simper interpreter operates on ten-digit decimal numerals (words), some of which it "understands" as legal instructions. Each operation mnemonic (Table II) corresponds to a two-digit code, ea h register has a one-digit code. The address field typically contains a three-digit memory-cell designator, or register and indirect address digits. The value in register P is always used as the memory address of the next instruction to be executed.)

Fig. 2. Structure of Simper's Simulated Machine

Table II

Simper Machine Operations

(* indicates operations added after the first experiment.)

Mnemonic	Action (if not obvious)
PUT	value of address field to register
LOAD	copy value in addressed cell into register
STORE	inverse of 'LOAD'
ADD	add value in addressed cell to register
SUBTRACT	
MULTIPLY	
DIVIDE	skip next instruction unless dividing by zero
DIVIDE*	set 'ERROR' flag on division by zero
LAND	decimal digit-wise minimum between register and memory
LOR	decimal digit-wise maximum
LEXOR*	"exclusive or": 'LOR' except for equal digits
JUMP	transfer to address if register is non-zero
JASK	transfer to address if a key has been typed
COMPARE	three-way skip on memory cell's value greater than, equal to, or less than register's value
SHIFT	
ROTATE	
EXCHANGE	flip contents of two registers
INCREMENT	
NEGATE	
ERROR*	overflow error code to register
ASK	decimal numeral from keyboard to register
WRITE	inverse of 'ASK'
CASK	ASCII character from keyboard to register
CWRITE	inverse of 'CASK'
IOT*	input/output transfer (for graphics etc.)
RANDOM	random 10-digit integer to register
TIME	seconds since midnight to register
WAIT	defer execution for milliseconds in register
HALT	stop execution
NOP	no-operation

Table III

Simper Interpreter Commands

(* indicates items added after the first experiment, ** indicates items added after the second experiment. Parenthesized phrases describe options obtained by terminating a command with the "altmode" key.)

Name	Action
DUMP	display decimal content of memory and registers (symbols too)
LIST or DEBUG	display memory content in assembly language (and machine language), 'DEBUG' shows "secret" tables
RUN	execute part or all of a program (and display registers)
GO	continue execution (and display registers)
CLEAR**	set a particular register's content to zero
FIX or EDIT*	change the content of one or more memory cells (and show prior content)
FLIP**	interchange the contents of two cells
SLIDE	relocate part or all of a program in memory
SCRATCH or ERASE**	erase all of a program erase all or part of a program*
FORGET or NAME	erase or attach a symbol to a memory cell (and say how much room remains for symbols)
NAMES	list all symbols and their cell associations (and their values)
SAVE or GET	copy memory to or from long-term storage
FIELDS	allow abbreviated instructions
FORBID or ALLOW	selectively alter the machine's instruction set**
NEWS	obtain the latest system news
HELP	obtain general information about Simper
control-G	stop any activity
GOODBYE or control-Z	log out

```
007 :LIST              (the user had created the following program)

001 :ASK A
002 :MUL A 10
003 :ADD A 6
004 :WRI A
005 :PUT P 1
006 :9
007 :
008 :
009 :
010 :2

007 :RUN$              (the user runs the program, "$" denotes altmode)

13:04:12               (the time)


EXECUTING 1 TO 500


P:      A:      B:      INSTR:
-----------------------------------------
1       0       0       ASK   A      INPUT NUMBER:4 ("4" typed by user)
2       4       0       MUL   A  10
3       8       0       ADD   A  6
4       17      0       WRI   A      NUMBER=17
5       17      0       PUT   P  1
1       17      0       ASK   A      INPUT NUMBER:0
2       0       0       MUL   A  10
3       0       0       ADD   A  6
4       9       0       WRI   A      NUMBER=9
5       9       0       PUT   P  1
1       9       0       ASK   A      INPUT NUMBER:-4
2       -4      0       MUL   A  10
3       -8      0       ADD   A  6
4       1       0       WRI   A      NUMBER=1
5       1       0       PUT   P  1
1       1       0       ASK   A      INPUT NUMBER:↑G (user aborts)

...15 INSTRS IN 1.100 SEC

007 :GO 4          (user continues a bit with no display)
2
13


...4 INSTRS IN .042 SEC

007 :EDIT 10
010 !3             (the user changes the function to be:  3x + 9)
007 :RUN
```

Fig. 3. Displaying a Simper Program's Activity

modified to communicate with various devices, including an "XY" plotter and graphic display terminals (the total system will be referred to as IMSSS Logo). A partial list of IMSSS Logo's primitive operations appears in Table IV, program editing/saving commands appear in Tables V, VI and VII, and the execution of a sample procedure is shown in Figure 4. The reader should maintain in mind that Logo is fundamentally a prefix language--commands may be composed of several operation calls, in which each operation is followed by a list of any arguments (possibly produced by other operations) it may need in order to be executed.

One of the few common aspects of the Simper and Logo languages is line-editing. Table VII shows the commands which allow users to correct typing and other errors before they terminate their command-lines (causing Logo or Simper to try to obey them). Particulary useful are the commands (control-E, -N and -S) which allow previously stored lines or words to be injected into the user's typing. One of the functions of good line-editing capabilities is to minimize the burdens on the poor typist.

Finally, it should be noted that in learning to use Simper, the student must learn the three languages (machine, assembly and interpreter) that are realized by the system. This is not a trivial matter for naive programmers, as the experiments have indicated.

Logo's interactive structure is more nearly unitary. Its basic piece of executable code is a line composed of one or more commands, and its basic piece of program (procedure definition), is an ordered series of lines. The Logo interpreter lways executing (or capable of executing) a user's commands, which may upon Logo primitives or the user's own procedures. Control returns to the user only when his or her last command and any commands it might have called have terminated normally or been aborted. A few of Logo's primitives may not be executed directly by a user's procedure, but there is not

Table IV

Some IMSSS Logo Primitives

(* indicates items added after the first experiment,
** indicates items added after the second experiment.)

Name	Action
TO	allows creation of a new operation (a procedure)
OUTPUT or RETURN* or REPLY**	allows operations to return values to the evaluator
EDIT	allows the user to change an operation's definition
MAKE	associates a name with a value
VALUE* or THING	accesses the value associated with a name
FRONT	moves the "turtle" or train forward
WHERE	returns the present location of the train
PLOT	sends turtle drawing to XY plotter or robot
SAY	causes the audio system to speak a message
PRINT	causes the user's terminal to type a message
REQUEST	asks the user for a message
SNAP	makes a "snapshot" of graphics picture being drawn
MOVESNAP*	moves a snapshot as part of an animated display
WORD	combines two words (of letters or numerals) into one
SENTENCE	combines two words or sentences into a sentence
FIRST	returns the first letter or word in a value
RANDOM	picks a digit between 0 and 9
SUM or ADD**	returns the sum of two numbers
IS or SAMEP*	are two words or sentences identical?
EQUALP	are two numbers equal?
IF THEN ELSE	decision making

Table V

Logo's Procedure Editing and Debugging Commands

Name	Action
TO	begin defining a new procedure
EDIT	begin modifying an existing procedure
TITLE	redefine the name of the procedure and its inputs
EDIT TITLE	change part of the title
LIST TITLE	display the title
EDIT LINE	change part of any line in the procedure
ERASE LINE	delete any line
LIST LINE	display any line
END	stop editing the procedure's definition
LIST	display any procedure's definition
ERASE	delete any procedure's definition or trace
ERASE ALL PROCEDURES	delete all definitions
LIST ALL PROCEDURES	display all definitions
LIST CONTENTS	display the titles of all defined procedures
LIST ALL ABBREVIATIONS	display the user's abbreviations for all operations
TRACE	display a procedure's arguments/returned value whenever it is executed
BREAK	halt execution (same as control-G)
EXIT	halt and print a message
GO	continue execution

(Indented commands may only be given after editing has been begun with 'TO' or 'EDIT'.)

Table VI

Logo's File-manipulation Commands

Name	Action
SAVE	replace an entry on a file with the current contents of memory
GET	append the content of an entry to memory
LIST FILE	display the entry names in a file
LIST ENTRY	display everything in an entry
LIST PROCEDURES	display only the procedures in an entry
LIST CONTENTS	display the titles of an entry's procedures
LIST ABBREVIATIONS	display the abbreviations in an entry
ERASE ENTRY	delete an entry from a file
COPY	copy a text file to or from a file entry

Table VII

Simper/Logo Line-editing Commands (* means Logo only.)

Name	Action
control-A or rubout	erases the previous character typed
control-W	erases the previous word typed
control-X	erases the whole line (also control-U in Simper)
control-R	retypes the present line minus deletions
linefeed	continues a line beyond 72 characters
return or altmode	terminates a line (altmode is also known as "escape" or "enter")
control-N*	insert (into the present line) the next word from the previous (or edited) line
control-S*	skip the next word from the previous (or edited) line
control-E*	insert everything remaining in the previous (or edited) line into the present line

(An example of Logo procedure editing:

```
←TO WELCOME
@10 SAY "HELLO THERE"
@EDIT LINE 10        (causes the line number "10" to be printed and
                      inserted into Logo's input buffer just as if
                      the user had typed it, so it may be erased.
                      Logo has now also grabbed the existing text of
                      line 10 and knows 'SAY' to be its first word)

@10 [ 01]20 ↑Nsay "↑S↑Nthere" [ "] GOES A WELCOME"

                      (The above editing line produced line 20 by
                      using line 10.  "↑" means "control-", Logo's
                      typing is in lower case, deleted characters are
                      in brackets)
@LIST
TO WELCOME
10 SAY "HELLO"
20 SAY "THERE GOES A WELCOME"  (the new line)

@)
```

```
←LIST' ACKERMAN (the user had defined the following procedure)

TO ACKERMAN :X: :Y:
10 IF EMPTYP :X: THEN RETURN WORD :X: "Y"
20 IF EMPTYP :Y: THEN RETURN ACKERMAN BUTFIRST :X: "Y"
30 RETURN ACKERMAN BUTFIRST :X: ACKERMAN :X: BUTFIRST :Y:
END

which realizes a string example of Ackerman's function)

TRACE ACKERMAN ('TRACE' will allow the user to follow ACKERMAN's
                    execution history, observing its arguments when it is
                    called and the values it returns when it is done.
                    Recursively generated copies of 'ACKERMAN' are
                    denoted by indentation)


←PRINT ACKERMAN "XX" "Y"          (execution begins)
ACKERMAN OF "XX" AND "Y"
  ACKERMAN OF "XX" AND ""
    ACKERMAN OF "X" AND "Y"
      ACKERMAN OF "X" AND ""      ("" is the empty string)
        ACKERMAN OF "" AND "Y"
        ACKERMAN RETURNS "YY"
      ACKERMAN RETURNS "YY"
      ACKERMAN OF "" AND "YY"
      ACKERMAN RETURNS "YYY"
    ACKERMAN RETURNS "YYY"
  ACKERMAN RETURNS "YYY"
  ACKERMAN OF "X" AND "YYY"
    ACKERMAN OF "X" AND "YY"
      ACKERMAN OF "X" AND "Y"
        ACKERMAN OF "X" AND ""
          ACKERMAN OF "" AND "Y"
          ACKERMAN RETURNS "YY"
        ACKERMAN RETURNS "YY"
        ACKERMAN OF "" AND "YY"
        ACKERMAN RETURNS "YYY"
      ACKERMAN RETURNS "YYY"
      ACKERMAN OF "" AND "YYY"
      ACKERMAN RETURNS "YYYY"
    ACKERMAN RETURNS "YYYY"
    ACKERMAN OF "" AND "YYYY"
    ACKERMAN RETURNS "YYYYY"
  ACKERMAN RETURNS "YYYYY"
ACKERMAN RETURNS "YYYYY"          (to PRINT)
YYYYY
←
```

Fig. 4. Tracing a Logo Procedure's Activity

a strict distinction between sets of commands as exists in Simper's three-level structure. However, a quirk in Logo's evaluation scheme imposes a different syntax on editing and management commands versus other operations. This will be discussed later. Readers interested in more detailed discussions of Simper and Logo should refer to report-1.

Peripheral Devices

Various terminals and controllable devices were available to Logo and Simper students during and after both experiments (Figure 5). The machine-language Logo interpreter was modified to dispatch graphics (or other special-device) commands to a Sail program: Sailogo (Figure 5). This program and Logo acted as coroutines. Hence, Logo's control of special devices was realized by Sail procedures.

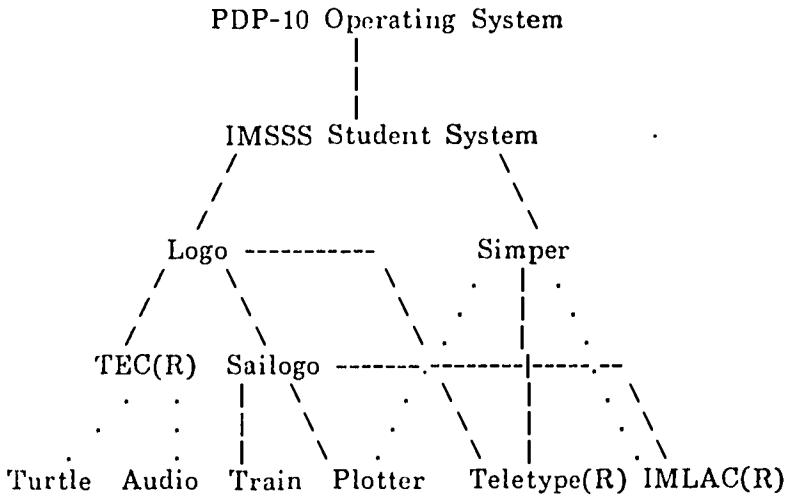
Most special devices played a relatively small role in the work reported on here. A brief summary of only the graphics and animation facilities will be included. All the devices are fully documented in report-1.[1]

Both experiments sought to establish good examples of how each device could be employed in solutions to programming problems. Some aspects of this will be discussed later with emphasis on relating device capabilities to teaching the concepts (e.g., those in Table I).

All the students whose work will be discussed here began their programming at conventional, model 33 Teletypes(R). This slow (10-characters-per-second), noisy, inexpensive but reliable terminal was their basic means for communicating with Logo and Simper until they had mastered the languages well enough to make good use of such special capabilities as graphics. In spite

---

[1] I am indebted to Steve Weyer for his fine implementation of the many special IMSSS-Logo features, such as graphics and animation



(Dotted lines mark connections made after the first experiment)

Fig. 5. Programming System Structure

of obvious drawbacks, Teletypes were in plentiful supply and provided paper printout for projects (like posters) whose results students wanted to take home. Some students retained a particular liking for Teletypes, because the mechanical bedlam generated by one in operation fascinated them.

Some fast (a few hundred characters-per-second), text-oriented, video displays were used occasionally by some students. These had limited, graphics-like capabilities, but they were not exploited in the experiments.

In the first experiment, two groups of students used IMLAC(R) PDS-1 graphics displays exclusively. These groups, however, are discussed only in report-1. For the students whose work is of interest here, the IMLAC displays constituted a goal, attained when a student's proficiency in the languages was adequate to allow comfortable use of the graphics system (Tables VIII and IX).

The graphics, line-drawing system emulates many abilities of the robot "turtle" developed at MIT and BBN (Feurzeig and Lukas, 1972b). It allows movement on the screen to be specified by "x,y" end-points in addition to the turtle's normal, roving-polar-coordinates scheme (in which movement is specified by 'FRONT' and 'BACK' along an angular heading changed by 'RIGHT' and 'LEFT'). For example, a square can be drawn by the Logo procedure 'SQUARE':

```
TO CORNER :SIZE:
10 FRONT :SIZE:
20 RIGHT 90
END
```

```
TO SQUARE :SIZE:
10 CORNER :SIZE:
20 CORNER :SIZE:
30 CORNER :SIZE:
40 CORNER :SIZE:
END
```

Table VIII

IMSSS Logo Turtle-Graphics Commands

Name	Action
CLEAR	erase the text area of the screen
WIPE	erase any drawing and put turtle home
SEE (HIDE)	make the turtle appear (disappear)
PENDOWN (PENUP)	enable turtle to draw visible (invisible) lines
PENP	return '"TRUE"' if turtle's pen is down, '"FALSE"' otherwise
POKE (UNPOKE)	stick out (pull in) turtle's head
HOME	move turtle to home position defined by 'SETTURTLE'
FRONT (BACK)	move turtle forward (backward) a specific distance
LEFT (RIGHT)	rotate turtle left (right) specific number of degrees
SETHEADING	point turtle on a specific angular heading
ASETX (ASETY)	move turtle horizontally (vertically) to an absolute screen position
ASETXY	move turtle horizontally and vertically to a position
RSETX (RSETY)	move turtle horizontally (vertically) a relative amount
RSETXY	move turtle relative to its present screen position
THERE	equivalent to an 'ASETXY' and a 'SETHEADING'
HERE	return turtle's current position and angular heading
ARC	make turtle draw an arc of specified radius and sense
ZAP (ZIP)	erase last turtle move(s) up to a visible line segment
PLOT (UNPLOT)	(do not) direct turtle commands to robot or plotter
SETSCALE	set screen resolution in units-per-inch
SETTURTLE	set both scale and home position on screen
WRAP	set up screen boundaries for wraparound
COMPRESS	shorten IMLAC display list (precludes use of 'ZAP' or 'ZIP')

Table IX

IMSSS Logo Animation Commands

Name	Action
SNAP	wipe screen and begin creating a numbered "snapshot" of whatever drawing (less erasures) is subsequently done
ENDSNAP	finish defining current snapshot and wipe screen
ERASESNAP	delete specified snap and its number
WHATSNAPS	return a sentence of currently used snapshot numbers
SHOWSNAP	display specified snapshot at turtle's screen position
PUTSNAP	identify a snapshot with an old or new "object" at a specific screen position, or move or erase an object
MOVESNAP	move an object (with wraparound) a relative distance on a relative heading and return object's final, absolute position ("R" in an object number has effect of 'RSETXY')
WIPESNAPS	wipe screen and erase all snapshots and objects

(A procedure for moving an object, referenced by a snapshot number, across the screen might be:

```
TO WALK :SNAPNUMBER:
10 SHOWSNAP :SNAPNUMBER:
20 ZAP      (a snapshot is a "line" under erasure)
30 FRONT 10
40 WALK :SNAPNUMBER:
END
```

or, better:

```
TO WALK :OBJECTNUMBER:
10 MOVESNAP :OBJECTNUMBER: "10 0"
20 WALK :OBJECTNUMBER:
END
```

for the latter, 'PUTSNAP' must first be used to tie a snapshot (an appearance) to an object at some screen position.)

Lines drawn may be erased by 'ZAP' and 'ZIP' commands, permitting limited picture editing as well as primitive animation. One student produced a short sequence showing a fuse "burning" down (disappearing into) and exploding a firecracker.

'PLOT' allows one to direct the effects of most graphics commands to either an HP7202A plotter or a robot turtle (General Turtle Inc., Cambridge, Mass.). Most students highly valued the ability to reproduce on paper what their programs had drawn on the display screens. Since students could use any type of terminal and still have their drawings appear on the plotter, this was exploited to encourage students to write and debug storable procedures rather than to just draw by direct commands. The plotter was only sporadically available during the first experiment and a true, robot turtle was available on occasion during both experiments. The robot came along with a "music-box" which was used significantly by two students in the second experiment.

During the first experiment, it became apparent that more powerful animation abilities would be possible and might serve as strong motivation for more complex student projects. Prior to the second experiment, genuine animation was added to Logo and Simper was modified to access the graphics system as well (Appendix 3, pages 1ST and 1LT). The 'SNAP' command allowed a student to save the effects of most subsequent graphics commands as a display subroutine within the IMLAC. These "snapshots" could then be shown anywhere on the IMLAC screen with 'SHOWSNAP' or 'PUTSNAP'. Snapshots of the same object in different orientations or sizes could then be shown successively in a "movie" (e.g., with 'MOVESNAP').

Although true animation ('PUTSNAP' and 'MOVESNAP' in Table IX) was not used by students in the first experiment, it was used in the second. Some students from the first experiment continued to work with Logo, influencing

some aspects of the developing animation system. A short film about Logo/IMLAC graphics and animation is available from IMSSS.[2] Figure 6 is adapted from that film.

Students used animation to produce such things as a flyable helicopter, a rocket launch, animated tic-tac-toe, movies of throbbing polygons, and a tennis-game. An example program appears in Appendix 4.

The computer could also be made to utter sounds (via the Logo primitive 'SAY') composed of any of several thousand prerecorded phrases, words, and phonemes stored in the IMSSS system. No organized use was made of this in the experiments, since it amounts to little more than the aural equivalent of 'PRINT'. Only a few terminals with audio output were available to students. Nevertheless, most students discovered the facility and some made imaginative use of it.

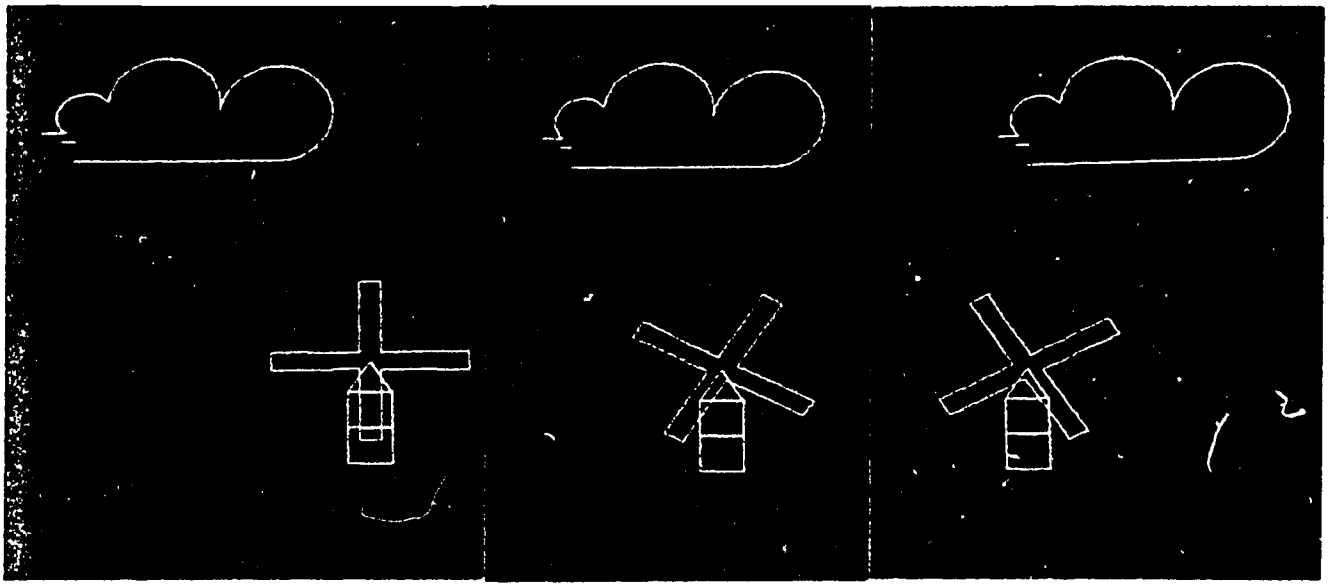


Fig. 6. Successive Frames from a Logo-Animation "Movie".

[2] Pat Crawley of the Stanford Communications Department produced this film, starring Adam Grosser, Greg Hinchliffe, Steve Spurlock, Steve Weyer and the author.

3 Students, Tutoring and Curricula

The desire to draw some conclusions about programming languages led to the student groupings shown in Table X. The first experiment had been concerned with assessing the value of graphics as well, accounting for the formation of groups IV and V. Those two groups are discussed only in report-1. Groups I, II and III in both the first and the second experiments provided most of the data for comparing the languages, evaluating the curricula and characterizing tutor-student-machine interactions.

The first experiment influenced many aspects of the second, some of which will be discussed here. For example, the enthusiasm generated by the graphics and animation system inspired the inclusion of graphics in late parts of both the Simper and Logo curricula, at a time when students had mastered either language "well enough".

Schools near Stanford were contacted in order to obtain inexperienced programmers, 10 to 15 years old--an age which is thought to ensure that children can master abstractions (Piaget, 1970).[1]

Teachers and others recommending students were asked not to base their selections on students' performances in school, because the intent was to study how any child learns to program. It had been observed previously that teachers tend to recommend only their better, mathematics students for such special projects. Apart from an admonition against such preference, the manner in which the invitation "to learn how to use a computer" was presented to students could not be controlled, so it cannot be stated that the enrollees constituted a cross-section of local students.

---

[1] I am indebted to Carolyn Stauffer for her invaluable help as liaison.

Table X

Experimental Groups

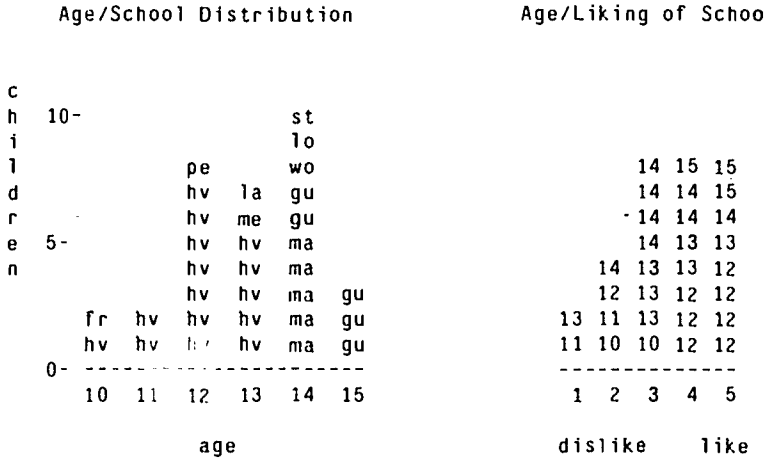
	Group	Composition
1973:	I	8 students learning Logo and then Simper
	II	8 students learning Simper and then Logo
	III	8 students learning Logo and Simper at once
	IV	5 students learning Logo with graphics
	V	10 paired students learning Logo with graphics
1974:	I	5 students learning Logo and then Simper
	II	5 students learning Simper and then Logo
	III	5 students learning Logo and Simper at once

In the second experiment, an additional source of "gifted" students was available. They worked at teletypewriters at home, were assigned to groups matching I, II and III, received the corresponding curricula on demand by mail and could call myself or others at Stanford for help during certain hours. Unfortunately, only a few of these students did significant amounts of work with Logo and Simper. Their work will be discussed at appropriate times, but these students are not indicated in the Tables and Figures.

More students responded than were needed for the groups outlined in Table X. As many as possible were accommodated, including friends who appeared later during the body of the experiments. Figure 7 presents some responses of the enrolling students to a brief questionnaire. Since students typically heard about the course from their mathematics teachers, the indicated preferences weren't surprising. As an aside, the students' attitudes toward school seemed to agree with observations in Jackson (1968) that one-fifth or more of all school-children will readily admit that they dislike school in general.

In all, about fifty students involved themselves in the first experiment, and correspondingly, about twenty enrolled in the second. To some degree, this insulated the experiments from the problem of dropouts. Transportation problems created a few defacto dropouts, particularly in the first experiment.

In the first experiment, students were scheduled to use the machine one hour per day, four days per week, with more regard for their convenience than for experimental grouping (Table X). Because the first experiment was in part a pilot study for the second, Fridays were reserved for modifying the curricula and debugging the interpreters or devices. However, on demand of some of the more interested students, Friday was considered open too.



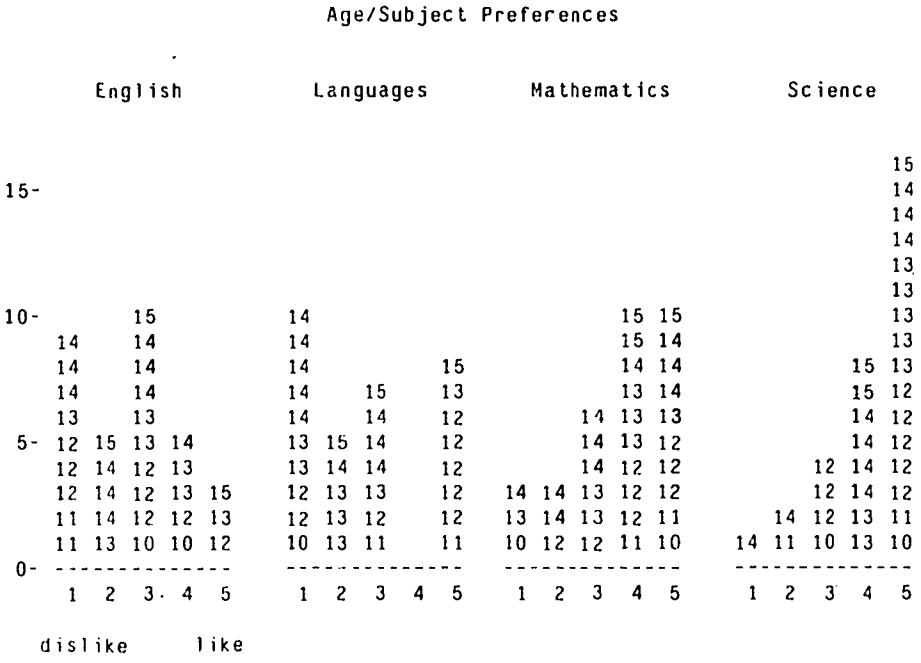
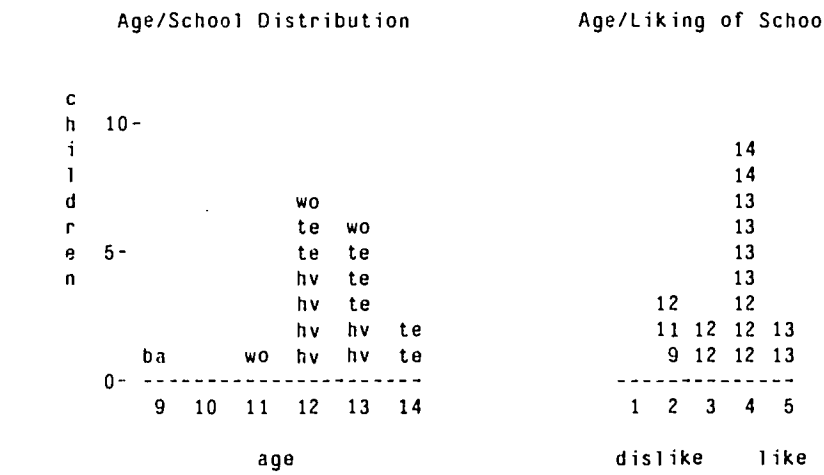


Fig. 7a. Some Information Characterizing the 1973 Students



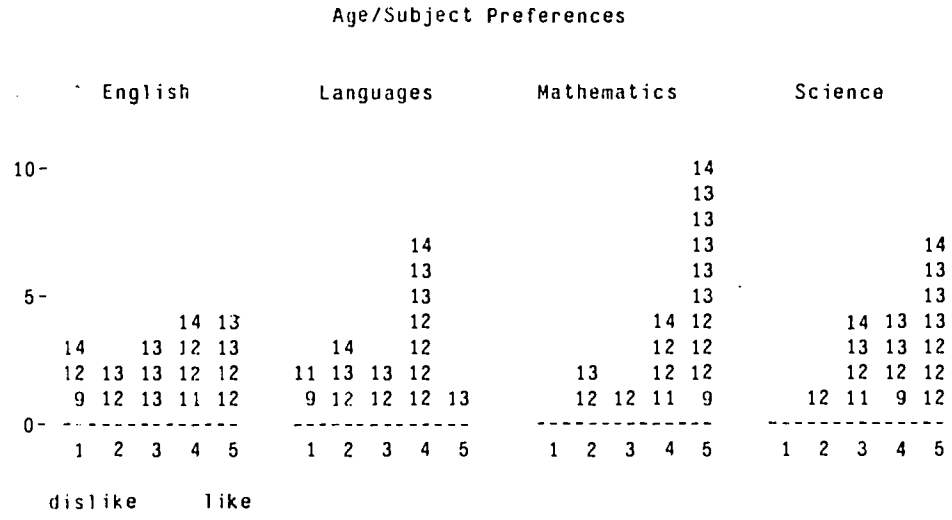


Fig. 7b. Some Information Characterizing the 1974 Students

In the second experiment, students used the machine three hours per week, on two-or three-day schedules. This was done because the first experiment had indicated that students should be segregated by group to allow more uniform tutoring and to minimize the inevitable distractions raised in a roomfull of students working at different places in different curricula or on projects in different languages. The students still retained the right to go to another room, after their scheduled session had ended, and use another terminal.

In order to obtain an initial assessment of each student's aptitude for programming, and to point out possible problems that each student might later have in learning the concepts, a test was constructed prior to the first experiment. It consisted of questions gleaned from a wide range of sources, because no one test in current use seemed to be valid for the range of concepts in Table I. A number of commercial programming tests were examined and some questions from these were used.[2]

However, all these tests relied heavily on timed sections of multiple-choice, often repetitious questions. Such structuring produces easily graded results and is commonly used to boost the "reliability" (correlation among test applications) of a test. In contrast, development of the test used for this work placed emphasis on the more elusive but crucial notion of validity, and on the exposure of thought processes (e.g., per Bloom & Broder, 1950).

A test, no matter how reliable, is utterly useless if it fails to measure the property of interest. It may even be dangerously misleading. In terms of the theory of testing and evaluation, as currently applied in the social sciences (e.g., see Worthen & Sanders, 1973; or, for the politics/realities of evaluation, see Jackson, 1968, and McLaughlin, 1974), validity like reliability is measured by

---

[2] Tests included: the ARCO Computer Programmer, the CPAB and Flanagan Industrial Test series by SRA, the ECPI data-processing test, and the IBM programming aptitude test.

correlative techniques. However, no matter how long the chain of correlations, validity is ultimately founded in human judgements and evaluations of quality. An example of validation taken from a commercial test-brochure is outlined in Appendix 2. It should alert the reader to some of the pitfalls that threaten those who wish to do aptitude testing, particularly with commercially available materials. Read critically, the example implies that testing theory and practice typically diverge when validity is demanded, yet validity of measures is precisely what must be demanded when meaningful research is the goal.

A test was presented to enrolling students for two purposes. One, some measure of the students' aptitude for learning the concepts was needed for matched grouping. Two, hopefully it would be possible to match the way students attacked particular questions in the test with particular aspects of their performance in the experiments. The test might therefore shed light on the tutorial needs of each student.

The preliminary test was constructed of some questions taken from the commercial tests mentioned earlier and questions of original design. All questions were formulated or reformulated to require constructive answers. The 1973 and 1974 tests are reproduced in report-1 and Appendix 2, respectively. Multiple-choice questions were thought useless. They force students to make judgements based on two levels: their relevant knowledge and the sensibility of the prescribed answers. The grader of such questions is freed of the burden of judging diverse answers simply by having it thrown onto the test constructer and the least-experienced judges: the students. What students think about each question and why they give their answers are important pieces of information that such testing destroys. For this work, students' answers were valued even if they were wrong or incomplete. Detailed answers would help evaluate the test as well as the students; and the judging would be done by persons experienced in the relevant fields (i.e., by the author or other programmers).

The desire for constructive answers to all questions on the test is best justified by those examples of "wrong" answers which nonetheless showed that students were thinking along the right lines. Figure 8 presents some for a question derived from a commercial test (note also the subtle defects in drawings B and C, and the beguiling A-B sequence). It is important to note that answers like those in the figure evidence approaches to the questions which would have been counted completely right or wrong if nonconstructive answers (e.g., multiple-choice) had been required. Figure 9 shows examples of totally unexpected answers to a question of original formulation.

One can neither assess a student fairly, nor know what a test is testing if the questioning scheme critically warps or limits information relevant to the purpose test.

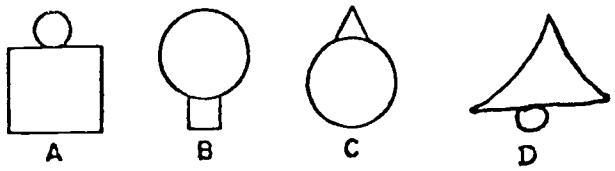
About one-hundred questions were selected for possible use in the test. Before the questions were presented to students enrolling in the first experiment, their difficulty, clarity, and the time required for their solution were evaluated by presenting the entire assemblage to several programmers (children and adults) in the IMSSS community.[3] As a result of this simple evaluation, most of the questions were accepted and were presented in two tests. Students answered one-third of the questions on the day they enrolled, being allowed one hour. The second test was to be completed at home at each student's convenience. The two parts of the test contained many similar questions. This was done because the preliminary evaluation had suggested that time should not be a factor in testing. Thorough and accurate evaluation of both test and students seemed to demand that as many questions as possible be answered. Two-part testing would also suggest whether or not any time limit should be applied to the single test which would be used in the second

---

[3] I am grateful to Marney Beard, Doug Danforth, Adele Goldberg, Paul Hechinger, Greg Hinchliffe and Lauri Kanerva for their help.

The Question and the Desired Answer:

Figure A was changed into Figure B by a simple rule. Please draw figure D so that it corresponds to figure C changed by the same rule.



What is the rule in words? BOTTOM SHRINKS, TOP GROWS

Other Answers:

TURN IT UPSIDE DOWN AND ALTERNATE SIZE

A IS A SQUARE WITH A CIRCLE, B IS JUST THE OPPOSITE

YOU CHANGE TO THE OPPOSITES

TAKE THE FIRST BASIC FIGURE AND CHANGE WITH THE SMALLER AND TURN UPSIDE DOWN

THE SMALL TOP FIGURE BECOMES LARGE AND THE OTHER BECOMES SMALL AND THEY TRADE PLACES

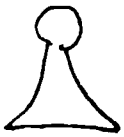


Fig. 8. Some "Wrong" Answers from the 1973 Preliminary Test

The Question and the Desired Answer:

What one rule, not using arithmetic, was used to make the digits on the right from the strings of digits on the left?

999999999	9
556	5
6106	6

TAKE THE FIRST DIGIT

Alternate, Unforeseen Answers:

THE DIGIT USED THE MOST

PREDOMINANT NUMBER

WHAT EVER NUMBER THERE IS MOST ON THE LEFT, PUT IT ON THE RIGHT

TAKE THE DIGIT WITH THE HIGHEST PLACE VALUE, OR THE ONE THAT REPEATS MOST OFTEN

Note: "number" was acceptable although "digit" or "numeral" were technically correct. More than half of the students who gave complete answers to this problem seemed not to be aware of the distinction. Their rank and choices of words contrasted as:

1973 or 1974 student rank	"digit" or "numeral"	"number"
at or above median	14	11
below median	3	13

Fig. 9. Some Novel Answers from the Preliminary Tests

experiment. Unfortunately, many of the students failed to complete the lengthy "take-home" portion of the test, either for lack of interest or because they dropped out. For the second experiment, it was decided that the test would be shorter and that new students would work on it during their first day, taking it home to finish if necessary.

Questions had been selected according to their apparent value in testing the ability to manipulate unfamiliar languages, model or analyze processes, form deductions, and visualize figural transformations (see Appendix 2). Some of the questions proved to be very useful for discriminating among the enrolling students. Two of these, the "candy-machine" and the "numbers-in-boxes" problems (Appendix 2, or report-1, page 169), required an understanding of concepts directly related to programming. Errors made by the students on these two questions were especially interesting and will be discussed.

In the candy-machine problem, a partial flow-diagram was provided in which few states had been left blank and connections between some states were missing. The task was to complete the diagram in any reasonable way. Many students had trouble with the basic idea that a process can be represented on paper as a diagram of the sequence of events in the process. They left blank states empty, filled them inappropriately, or misconnected the dangling states. Errors in the solutions given could be divided into three classes: (1) assignment of unreasonable destinations for unconnected arrows, (2) assignment of unreasonable functions for undescribed states, and (3) treatment of the entire diagram as a maze in which only one path was to be marked as a likely protocol. Errors in class (1) or (2) suggest that a student had trouble using the information already present in the diagram to deduce reasonable "things to do next" or "things to do now". Class (3) is interesting because such errors indicate that a student viewed the diagram as a menu of instructions from which to choose one plausible sequence, rather than as a complete description of all possible sequences, for some process.

The numbers-in-boxes question asked the students to obey a short, program-like sequence of arithmetic instructions which operated on some numbers written in a set of numbered boxes. Very few students correctly obeyed the instruction which read: "Add the number in box 7 to the number found in the box whose box number is in box 6, and write the sum in box 6". The sentence is hard to read, but the idea that a number (value) in a box could be used as the number (name) of a box (indirect addressing) was the typical difficulty. Many students also had trouble with the idea that writing a new number into a box should destroy its previous contents. Solutions fell into a few distinct classes which can be attributed to failures in the understanding of those two concepts.

In both experiments, the test was used to establish a rank ordering of enrolling students, and performance on the test seemed to break into a few levels. For the first experiment, roughly equal numbers of students from each level were assigned to groups I, II and III. The second experiment's grouping was more constrained by the interaction of students' scheduling preferences with the desire to keep the groups in separate time-slots. In both experiments students determined their own class schedule within the time constraints mentioned earlier.

Figure 10 shows the composition of the groups according to testing rank, age and amount of time spent in actual work with the interpreters. The candy-machine and the "logic" (Appendix 2, or report-1, page 170) problems tended to be most influential in discriminating among students of equal age above and below the median. The youngest had the most trouble with the candy machine. They missed the point that the diagram was an overall description of the machine. A few of the older students were familiar with flow-charts from school and thought that problem easy. In the first experiment, they had also been students who enrolled late. These late arrivals usually did very well

	Group	Age	Hours Spent Using Logo & Simper	
	III*$	15	36.4	
	I$	15	35.7	
	II*$	15	18.1	
	III$	13	29.7	
	III*	13	11.1	
	I*$	12	12.7	
	II*$.....	13	59.2	
	III#	14	0	
	I#	14	5.8	
	II.....	13	50.6	
	II	14	33.5	
	II	12	22.8	
	II#	14	6.4	* marks students who enrolled
	II*#	14	27.9	late.
	I#	14	5.7	
median..	I#	14	5.7	# marks early dropouts.
	II#	14	0	
	I	11	24.2	. marks significant breaks in
	II#	14	4.9	performance on the test.
	III	11	14.2	
	I	12	23.0	$ marks those who continued
	I	14	18.0	programming well beyond the
	III*#	13	2.1	experiment.
	III	12	11.0	
	III	13	28.7	
	I$	12	27.1	
	I*	10	11.7	
	II.....	12	21.1	
	III	12	20.1	
	I	12	19.1	
	III$	10	35.7	
	II	13	6.1	

Fig. 10a. Student Ranking on the 1973 Preliminary Test

	Group	Age	Hours Spent Using Logo & Simper	
	I$	12	24.2	
	III$	9	47.8	
	III*	13	27.1	
	III	12	33.2	
	II	13	15.4	
	II	13	26.9	* marks students who enrolled
	II.....	13	25.9	late.
	I*	12	26.3	
median..	II$	13	33.7	# marks early dropouts.
	II	11	14.3	
	I$	14	45.4	. marks significant breaks in
	I#	13	3.2	performance on the test.
	III.....	14	12.0	
	III	12	38.6	$ marks those who continued
	III*.....	12	16.2	programming well beyond the
	I#	12	1.5	experiment.
	I	12	8.0	

Fig. 10b. Student Ranking on the 1974 Preliminary Test

with the test, perhaps in part because they worked on it quietly alone--a feature lacking in the massed testing of the first enrollees. This provided another reason for eliminating timing of the test in the second experiment.

Examining the first experiment's test-results in terms of four constituents: the first three problems mentioned above, and everything else, the students' performances compare generally as follows. Students at the bottom of the ranking (Figure 10a) were unable to grasp the candy-machine and the box-program questions, they correctly analyzed only the clearest statements in the logic problem, and they failed to finish the test by a large amount. Students near the middle filled only the empty states in the candy machine reasonably; they correctly obeyed all commands but the indirect-addressing command in the box program, with some failures to erase a box's content when they wrote into it; they only missed the fourth statement in the logic problem; and they did fairly well on the rest of the test, though not always finishing it. Students near the top correctly filled all states and connected all the dangling arrows in the candy machine, a few of them missed the indirect-addressing command in the box program, they did the logic problem correctly, and they typically finished the rest of the test. Similar comments apply to test results for the second experiment, with the qualification that these students seemed to do better on the test than did those in the first experiment.

Of course these breakdowns are not rigid. In particular, it is very hard to order many of the tests in the broad middle regions of the rankings. Ranking forces transitivity upon performance ratings for solutions and problems which are often qualitatively different. But by demanding constructive answers, the answers contained much detailed information about the students and the test. If the test had been an exercise in multiple-choice, it is not clear what information it would have conveyed, but it certainly would have conveyed less.

Some changes in the test were made as a result of the first experiment. Aside from making it shorter and unitary, and applying it individually with no time limit, changes typically involved readability and the elimination of frivolous questions.

Tutoring. Both experiments were planned to depend upon written curricula which would control the basic information given to students. Interpreters for the programming languages would simply act as computational resources which the students could use to work problems in the curricula or experiment with on their own. However, any attempt to develop a fully self-contained curriculum for programming was deemed unrealistic. The main concern was gaining access to tutorial protocols generated by novice programmers working in the best possible environment for learning. Therefore, human tutors were provided who could help students over failures in the curricula and report their interactions. The tutors were to be knowledgeable in the programming languages being taught and would be familiar with the corresponding curricula.

In the first experiment, it was hoped that enough tutors would be available each day to guarantee at least one for each five students in each group.[4] Two instructions to the tutors were emphasized: (1) never type anything for the student on his or her own terminal, even when giving the most direct help, all typing must be the student's; and (2) when asked for help on any problem, encourage the student to formulate and try out his or her own ideas first, before making other suggestions. It was hoped that these instructions would guarantee the purity of the protocol data and help the students to think as much about generating and debugging ideas as about getting correct results.

Unfortunately, this tutoring effort failed in some crucial functions. First,

---

[4]My thanks go to Avron Barr, Marney Beard, Doug Danforth, Adele Goldberg, David Rogosa and John Shoch for their help as tutors.

initial enthusiasm faded quickly and most tutors became sporadic in making their scheduled appearances. This seemed largely due to their lack of prior experience in working closely with, and at the immediate demand of, several children at once. Second, and accordingly, the tutors could not maintain detailed logs of their interactions. Third, the tutors did not always keep up with new developments in the curricula, partly because its production fell behind the students' pace and partly because pieces of it were designed "on the fly" to patch mistakes/omissions. In either case, new curriculum-text was made available to students and tutors simultaneously--a bad policy.

Therefore, for the second experiment, tutoring was to be done by one person (the author) working with at most five students, all in the same group (per Table X), with the appropriate curriculum ready well in advance of each session. This facilitated note taking, gave the students personal, more uniform help, and ensured that problems with the curricula/interpreters were caught quickly. It is one reason why the number of students in the second experiment is smaller than it was in the first.

Curricula. Development of "parallel" curricula for Simper and Logo proved to be the most demanding task in setting up the experiments. Both the concepts and the languages had to be taught, and this is done best with example problems, some of whose solutions students must copy, modify or generate. The ability to teach both the concepts and the languages would be very sensitive to the choice of problems. For the students, the experiment was to serve to improve their literacy on the subject of computers and computation. Again the choice of examples and projects would be important.

Unfortunately documentation of problems used in similar work by others was scarce or cursory. Furthermore, most of the relevant research had been based on Logo or an equivalent high-level language. Problems appropriate for a

low-level language such as Simper are typically quite different. That was the fundamental obstacle to achieving apparent parallelism, given the intentionally diverse natures of the languages to be taught. So, the curricula were constructed to teach the concepts in roughly the same order, using whatever features each language possessed that could best be exploited for each concept.

As well as the concepts, the mechanical details of each language had to be taught. A few features (line-editing, Table VII) of Simper and Logo are very similar and were taught at the same time in the same way. But most features were taught differently, either because they were appropriate to different concepts or because they were needed at different times as tools in the general structure of each language. The Logo and Simper curricula are documented, as they were during the first experiment, in report-1. The discussion here will concentrate on the changes to the curricula which resulted from that experiment, in preparation for the second (see also Appendix 3).

Each curriculum was divided into five logical parts, each typically discussing more than one concept. Each part gave students programs to work on and fill-in-the-blanks questions to answer. The parts were distributed one at a time, giving the author a chance to review each student's work on them. Those students learning Simper and Logo simultaneously (group III) alternately received parts for each language.

The concepts were presented only very roughly in the order of Table I. For instance, the concept of a heuristic was introduced relatively early via a scheme for thinking about recursive algorithms. This involved a brief case analysis of some problems (derived from Polya, 1957): (a) what case can be computed? (b) how do I detect that case? (c) if not that case, then how do I generate one closer to it? (d) what must I remember for each case? and (e) when do I stop? In procedural terms, (a) and (b) form the procedure body, (c) is the recursive step, (d) preserves local context, and (e) is the stopping rule.

A special effort was made to produce visually pleasing curricula. Path pointers gave direction to the student, making the next question or instruction contingent upon the student's latest response. This subtly introduced decision making and sequencing (program control). It was, however, a bit too subtle for most students. Cartoons and examples were chosen for humorous as well as conceptual merit, and summaries were included so that the curricula could endure as reference material.

Changes in the curricula between the two experiments centered on reordering and reformulation of discussions of several concepts. One effect was reduction of the sizes of both the curricula to roughly sixty pages (a reduction of 1/3 for Simper and 1/5 for Logo).

Modifications to the Simper curriculum were based upon apparent student confusions in the first experiment. In Part 2, an explicit reminder was added as to why computers don't understand human languages (because humans themselves have yet to comprehend their own faculties). This helped to clarify the curious results students obtained when they followed the advice to "type anything you please". Otherwise, Parts 1 and 2 remained unchanged (see report-1). Parts 3, 4 and 5 then proceeded along a mostly new course in covering material previously allocated a dozen parts.

The new approach hinged on teaching machine-language first and thereby motivating both the desirability of the more convenient assembly-language and the need for the interpreter's powerful editing language. All this was permeated with allusions to message processing and computational context. The former being a metaphor used with some success in tutoring experiment-1 students, and the latter being an essential concept that had been troublesome to many of those students.

Part 3 first sought to clear up the lesser problem of what literals are in

the language by demonstrating more examples. It also tried to motivate the need for registers as a scratch pad. It then approached one aspect of context: attention. The machine was described as giving its "attention" to registers and memory cells when in the process of executing a program--only certain values in those cells could be "understood" as legal instructions. Without an ability to focus its attention on a source of messages, the machine would be quite useless. Registers, as defined by the machine's structure, were described as a means for passing messages between instructions, reflecting an aspect of the machine's internal context. In spite of the simplicity of the machine-language programs written in Part 3, editing commands such as 'SLIDE' (Table III) found direct application; and a few students suggested new ones (e.g., 'FLIP').

Simper Part 4 reviewed two of the three segments of machine-language instructions covered in Part 3 (i.e., the operation and register fields), and went on to motivate the need for the address field as a source of the second input to binary operations (e.g., addition) and as a means for accessing "full-word" chunks of data. Since the structure of most machines modelled by Simper was once dictated by both technology and economics, a brief word to that effect was included in the tutoring. The essential role of memory in any machine deserving of the name "computer" was alluded to. Using a time-telling program developed in this and the previous part, students were led into assembly language. The use of new editing commands (e.g., 'LIST'), designed especially for this second language, were also introduced. The remainder of Part 4 dealt with execution sequencing, and decision making. It attempted to motivate these with an odd/even number-testing program analogous to one used in the Logo curriculum. This problem was formulated as a test of the student's ability to translate an English statement of a program into Simper. Students having trouble writing the program were helped, and details of this tutoring were recorded. The final version of the program demanded an understanding of

literals, names (in the form of machine addresses), binary operations, register and memory-cell intercommunication, conditional and unconditional branching, and the communication of symbols to and from the typist. To cap off this work and prove that problems can often be solved in several externally equivalent ways, the curriculum suggested rewriting the program with fewer instructions (three basic forms existed).

Simper Part 5 attempted to crystallize the idea that interactive programs define new languages and thus set up new contexts when run. Student-defined symbols (names) and relative addressing were introduced as conveniences, peculiar not just to assembly-language programming. They found application in a random-number, guessing-game program used also in the Logo curriculum. The decision-making operation ('COMPARE') was then introduced as a way of making the students' programs smarter--they could now give their users hints like: "GUESS HIGHER". At this point, the concept of a function was introduced much as in the original curriculum and with the same visual aids. Part 5 closed with some reviews of messages and context in terms of the "domains" of functions.

Now students could go on to learn how to use the Simper graphics capabilities (Section 2), which were identical in power to those of Logo. They could also begin to learn Logo if they had not already. As in the first experiment, most students did not complete both curricula, so things like "pushdown stacks" were discussed only in terms of special projects which a few students undertook.

Part 2 of the new Logo curriculum was changed in the same way as was Simper Part 2. Part 3 kept the old discussions of literals and simple, direct commands, but then led into procedures as program elements, rather than naming (e.g., with 'MAKE'). Time- and date-telling procedures were the focus

because experiment-1 students had generated these on their own and had found them useful as well as instructive. The idea that problem solutions could be broken into logical parts was demonstrated simply by a procedure that called both the time and date procedures.

Logo Part 4 discussed naming first in terms of procedure names and then in terms of input variables to procedures. 'MAKE' was only introduced when a student's special project absolutely demanded it. Execution control was illustrated in terms of a procedure that called itself unconditionally, running forever. This was parallel to what had been done in Simper. Editing commands were reviewed, and message passing and context were developed in terms of procedure inputs. Block diagrams, which had little success in the first experiment, were simplified and given a second chance as aids. As for Simper, the introduction of functions was unchanged. Logo's parsing of complicated command-lines was depicted with diagrams, and a fill-in-the-blanks script adapted from the original curriculum.

Logo Part 5 opened with decision-making as an essential ability of any true computer and a brief discussion of the programmer's role in using such abilities for his or her purposes. The various Logo predicates were covered using block diagrams, examples and exercises taken from the original curriculum. The part predicates play in decision-making was emphasized. Composition of commands was discussed, particularly along the lines of a telephone-call metaphor. This was expanded further in terms of good program articulation as the following program was developed. The use of simple recursion (iteration) and stopping rules was motivated by a clock simulator which printed "TICK" or "TOCK" depending upon whether the time (in seconds) maintained by the system was even or odd. Applications of Logo's two decision-structures ('IF'... and 'TEST'...) also were contrasted with this program as:

```
TO KLOK                    TO KLOK
10 TEST EVENP SECONDS      10 IF EVENP SECONDS THEN PRINT "TICK"
20 IFTRUE PRINT "TICK"        ELSE PRINT "TOCK"
30 IFFALSE PRINT "TOCK"    20 KLOK
40 KLOK
```

where

```
TO SECONDS
10 RETURN BUTLAST BUTFIRST BUTFIRST TIME
```

and

```
TO EVENP :X:
10 RETURN ZEROP REMAINDER QUOTIENT :X: 2
```

were also defined by the students. That Logo has, as most languages have, redundant operations, was demonstrated by having students write a procedure ('AIN'T') equivalent to 'NOT'.

True recursion (making use of local contexts) was introduced, as in the earlier curriculum, using the "little brothers" analogy of Brown and Rubinstein. The true effect of returning control but not a value from a recursively called instance of a procedure was clarified.

The concept of a "bug" (unforseen error) in a program was illustrated by a number-guessing-game program similar to that in the Simper curriculum. Students were asked to design the program and then modify it in several ways, all of which, except the last (using 'COMPARE'), suffered from particular inabilities to interact reasonably with the human guesser:

```
TO QUIZ :PICK: :GUESS:
10 TEST COMPARE :PICK: :GUESS:
20 IFTRUE PRINT "SMARTY!"
30 IFTRUE QUIZ RANDOM REQUEST
40 IFFALSE IF LESSP :PICK: :GUESS: THEN PRINT "GUESS LOWER"
                                  ELSE PRINT "GUESS HIGHER"
50 IFFALSE QUIZ :PICK: REQUEST


TO COMPARE :X: :Y:
10 IF BOTH NUMBERP :X: NUMBERP :Y: THEN RETURN EQUALP :X: :Y:
                                   ELSE RETURN "FALSE"
```

Logo's file system was introduced at this point because it seemed natural that students would want to save this particular program. As had been done earlier for editing commands, a one-page manual was included for file manipulations and abbreviations. Some examples gave practice.

Recursive procedures that return values were introduced using a more thorough treatment of an example from the original curriculum. Both block and little-brother diagrams were used to describe how a procedure that removes all instances of a selected letter from a selected word should work. The students were asked to try their hands (and heads) at solving the problem by synthesizing the procedure. Errors and questions were to be noted and solutions were provided. A playlike script attempted to solidify understanding of one solution. Different forms of solutions (e.g., left- and right-recursive) were also discussed. Then a modification was suggested which would lead to the solution of another problem: writing a procedure to reverse a word. Up to this time, no stopping rules had been concerned with numerical criteria. Now, counting and program self-modification were introduced by a procedure that counted up (or down) to a limit and then modified itself permanently by self-erasure:

```
TO SELFDESTRUCT :HOWSOON:
10 IF LESSP :HOWSOON: 1 THEN ERASE SELFDESTRUCT
                        ELSE SELFDESTRUCT DIFFERENCE :HOWSOON: 1
```

The final of Part 5 developed Polya's ideas on solving problems in terms of the structure of general recursive procedures. Several projects derived from the first curriculum were presented. Students could then go on to Simper and graphics, as they pleased.

The graphics curriculum was derived from that presented to students in groups IV and V (Table X) in the first experiment. Since students in the second experiment would have already mastered much of the basic languages, it was shortened (to 7 pages) and concentrated on animation projects. Each

student who completed either or both languages was asked to think of a project to work on, graphics providing an enjoyable and quite acceptable medium.

What and how the students were taught were functions of two main beliefs: (a) testing should be an educational experience; and (b) people should understand as many of the valuable products of their culture as possible.

The nature of the tutoring prescribed implicit testing of each student, yet all students always got each "answer" eventually. Especially in the second experiment, students saw their actions precipitate prompt, accurate tutorial responses. For both the students and the research, a working goal was to have students come to feel at ease with dialectical responses to their questions. For a few students, this proved to be a difficult departure from their accustomed experiences in formal schooling.

One knows not when a cultural product might be essential (physically or psychically) to the individual or to the whole culture. But value is subjective and evanescent, and one who finds an application for a cultural artifact may not also find others expressing agreement that the application is valuable. Nevertheless, any successful try at an application (discounting plain luck) first demands some understanding. This reeks of technology, yet art, history, engineering and gastronomics all draw from science to form their own technologies. In short, everyone should understand and be comfortable with his or her machines (e.g., Pirsig, 1974)--in the particular instance here, the "machine's machine": the computer. Some of the children in these experiments hopefully would benefit in just this way, even if they might not discover the fact for years. A nagging fear that this might be a vain hope was instilled in this author when conversing on this research with a successful educational researcher, who regularly uses computers for statistical analyses. Hearing that computers can do more than perform numerical computations left that professional surprise an example of how a tool can be misunderstood.

4 Data Acquisition and Analysis

The simple methods chosen for obtaining data and the type of analysis believed to be appropriate for this essentially qualitative study will be discussed here. Some reasons why the analysis should not be founded naively upon classical statistical inference will also be outlined here.

Throughout both experiments, the Simper and Logo interpreters saved information on each student's activities. Each command or response typed by a student was appended to his or her individual protocol file on the operating system's disc-storage. Prompts and error messages elicited from the interpreters, and output from students' programs were also saved as they happened. Each such piece of information was tagged with its time of occurrence. At the end of the first experiment, the Logo and Simper interpreters were modified to accept these files directly, in place of keyboard input. Each student's interactions with the interpreters could thus be replayed and be observed in their proper context. In addition, the error-message and timing data in the protocol files could be analyzed in more conventional ways by forming summary statistics such as error frequencies and typing delays (response latencies). This sort of data was not of particular interest, except insofar as it could be used to point out particularly common errors, or confusions due to imperfections in the curricula or the tutoring. Some additional data were obtained from notes made by the tutors during the first experiment and the author's notes from the second experiment. The bulk of the data derives from the latter notes and replays of recorded protocols. Some problems with the IMSSS time-sharing system, encountered during the first experiment, are discussed in report-1. Most of them also affected the second experiment in minor ways.

The usefulness of these experiments rests upon the ability to understand students as they have tried to learn Logo, Simper and the concepts explained in

the curricula. Classical hypothesis-testing is not of concern in this wo ., although others have attempted to reduce their analyses of children learning programming to clinical forms, e.g.:

> *"Children who have had a Logo experience for several semesters will perform significantly better on problem solving tasks than children who have been in a non-Logo control environment."*
>
> -- Folk et alia, (1973).

For this work, the goal has been an exposure of basic features of how children think in the relatively unconstrained environment of a programming laboratory. That is a qualitative exercise in careful judgement, and it centers on a detailed study of errors made by students as they try out new ideas for themselves. But, as in any analysis of data, an analysis of errors must be valid in the sense that its meaning is not warped by analytical constraints.

> *"It is a capital mistake to theorize before one has data. Insensibly one begins to twist facts to suit theories, instead of theories to suit facts."*
>
> --Sherlock Holmes, by Sir Arthur Conan Doyle.

Whenever statistical procedures (such as classical hypothesis-testing) are applied to data, certain mathematical assumptions (e.g., of scale and distribution) about the data must legitimately be met, if resulting conclusions are to carry any scientific weight. In too many research settings, the importance of procedural assumptions is ignored, generating technically invalid or misleading analyses.

These remarks evangelize to those who, perhaps as students or other well-intentioned researchers, might be seduced by the apparent power or elegance of various, common, analytical procedures (e.g., analysis of variance), while being unaware of some of their potential for frivolous application to expediently massaged (e.g., vacuously scaled, "transgenerated" and/or "Windsorized") data.

In the social sciences, especially in education, the style of research too often reflects a Quixotic quest for numerical results, apparently stemming from the belief that quantitativeness is a precursor of objectivity and respectability in one's discipline.

> *"They use statistics as a drunkard uses lampposts, for support rather than illumination."*
>
> --Andrew Lang.

For instance, some psychologist's fundamentally qualititative data might mysteriously be provided a "scale" on which important "variables" could be "measured"--the accrueing benefit to psychology ranking with that brought to music by some quitarist's chance strumming of the Lost Chord. Quantitativeness at any cost is a precursor of sham not objectivity. This, and the dangers lurking in the fog of "cookbook" mastery of statistics, are amplified by the relatively easy access most researchers now have to computerized, statistical procedures (e.g., Ellis, 1972). Perhaps as seriously, widespread use of standardized procedures has led to stereotyped theorizing (e.g., to hypothesis testing restricted to linear models and Gaussian-distribution theory), wherein convenient rather than reasonable procedures define the theory, and the implicit necessary assumptions of the procedures are virtually ignored. The judgemental analysis for this work hopefully respects the qualitative nature of the data to which it is applied.

An example taken from Simper protocol data illustrates the nature of the judgemental analysis used here. It shows how one student suddenly seemed to grasp a concept with which he had been having trouble--name-value association (addressing) in Simper. If the programming is unclear, the reader should refer back to Chapter 2. The student's dialog with Simper is reproduced here as he was engaged in writing a program to realize the function: $x^2 - 3$:

```
003 :2

015 :ASK A
016 :STORE A 200
017 :MULTIPLY A A
018 :SUBTRACT A 3
019 :WRITE A
020 :RUN 15:
```

He appears to understand the purpose of addressing in 'STORE A 200', but his program contains several errors that suggest otherwise. The first causes execution to stop at 017 because the symbol 'A', used in the address field of the instruction in 017, has no binding and thus no associated value. The student thought he could square the A register's content with the instruction: 'MULTIPLY A A', and he thought he could subtract 3 from that with: 'SUBTRACT A 3'. In both cases, the meaning of the register field seems to be understood, but the address field is misunderstood. The student corrects the first error (messages from the interpreter are in lower-case):

```
020 :FIX 17
017 :MULTIPLY 200 200
200 isn't a register, use a, b, or p
017 :MULTIPLY A 200
020 :RUN 15:
```

and the prog[illegible]works except that, because location 3 contains the value 2, the subtraction [illegible]n't do what he expected. At this point he seems to understand that he can store and access values via addresses (names) because of his correct use of the register and address fields of the 'STORE' and 'SUBTRACT' instructions. But the idea crystallizes:

```
020 :FIX 201
201 :3
```

when he associates the desired value 3 with the name (location) 201,

```
020 :FIX 18
018 :SUBTRACT A 201
```

and correctly accesses it to complete his program. From this dialog, one can see the student begin to apply the concept in correct fashion (in the 'STORE' instruction), then fail because he has not yet mastered it fully, and finally succeed, partly helped by simple error diagnostics. The student later made a similar mistake, but corrected it at once.

For the purposes of these experiments, this type of analysis can suggest when and how a student masters something presented in the curricula. Students can be compared in far greater detail than can be done with occasional discrete tests, the curricula and languages may be evaluated very finely, and the preliminary aptitude test's validity may be rated subjectively.

The language evaluation aspect of the protocol analysis is partly demonstrated by the following examples from Logo and Simper protocols of absurd or misleading responses to students' syntactic errors. First, consider:

```
←PRINT :::SNOOPY:::
don't use the empty thing for a name
```

in which the student's obvious attempt at multiple indirect-addressing is completely misconstrued by Logo's simplistic parsing (the first pair of colons are found to contain no name string). And second:

```
001 :SUBTRACT 1 FROM P
002 :RUN
warning! you forgot to name a location fromp
illegal memory reference 0 at 1
```

in which Simper, striving to extract three fields and no more from the student's line, compressed a simple syntactic error and generated a more advanced type of error. Not only was this spurious error unrelated to what the student had done, it exposed the student to a situation for which he was not yet prepared (i.e., the use of assembler symbols). These examples were taken from the first experiment's data. Since it was in part a pilot study for the second, the

analysis led to changes in Logo and Simper that corrected at least some of these kinds of faults.

It should be clear that the Logo and Simper interpreters used are not "smart". They do not tutor their users on the semantics of programs--in the experiments, that was left to humans. The interpreters do little more than trap syntactic errors, sometimes acceptably well:

```
001 :SHIFT
unspecified register, use a, b, or p
001 :SHIFT 76
76 isn't a register, use a, b, or p
001 :SHIFT A
shift uses l, or r or @ and a number in the address field
001 :SHIFT @56
@56 isn't a register, use a, b, or p
001 :SHIFT L 56
l isn't a register, use a, b, or p
001 :SHIFT A L57
```

As was mentioned earlier, a simple analysis of the protocol files was also carried out. For example, if a Simper student's errors were categorized and plotted as in the graph in Figure 11, an interesting effect usually could be observed: familiarization with the language led to a decrease in errors classed as syntactic and an increase in those classed as semantic--an infererence being that as students increase their active programming vocabulary, they can more easily realize their ideas about problems as programs and find that their ideas (now programs) aren't always debugged. But this is more reasonably corroborated by tutorial data and detailed protocol analysis.

The tutoring process often was dialectical, especially when students became confused. It therefore possessed an analytical facet which influenced the recorded data. For example, when students expressed doubt about their ability to solve a particular problem, they were asked to explain the solution they had attempted, then they and the tutor examined the pros and cons of this in relation to the problem statement, converging toward a working solution. For

```
bill163.dta;3  AUGUST 1, 1973 12:20PM

1 DAYS, 1 LOGINS, 33.40 MINUTES ON, 372 KEYS TYPED ON 60 LINES.

RESPONSE DELAY, MEAN & DEVIATION:  32.15   74.36 SEC.

 1.00 LOGINS/DAY, 33.40 MINUTES/DAY, 372.00 KEYS/DAY

 33.40 MINUTES/LOGIN, 372.00 KEYS/LOGIN, 60.00 LINES/LOGIN

 11.14 KEYS/MINUTE, 6.20 KEYS/LINE, 1.80 LINES/MINUTE

36 ERRORS:  36 GENERAL, 0 NAME, 0 RUN, 0 FIXUPS

36 SYNTAX ERRORS, .60 SYNTAX ERRORS/LINE, 1.08 SYNTAX ERRORS/MINUTE

 .00 RUN ERRORS/LINE, .00 RUN ERR    'MINUTE

  0            10        20        30        40        50        60
  |    .     |    .    |    .    |    .    |    .    |    .    |    .
 1|####################
 2|##########
 3|##
 4|#
 5|#
 6|#
 7|#
 8|#

 1|UNSPECIFIED REGISTER, USE A, B, OR P
 2|EMPTY ADDRESS FIELD?
 3|SHIFT & ROTATE USE L, R OR @ & A NUMBER IN THE ADDRESS FIELD
 4|EXCHANGE USES A REGISTER IN THE ADDRESS FIELD
 5|ONLY VALUES FROM 0 TO 999 MAY BE PUT
 6|ISN'T A REGISTER, USE A, B, OR P
 7|SHIFTS OR ROTATES MUST BE BETWEEN -999 & +999
 8|UNKNOWN OPERATION
```

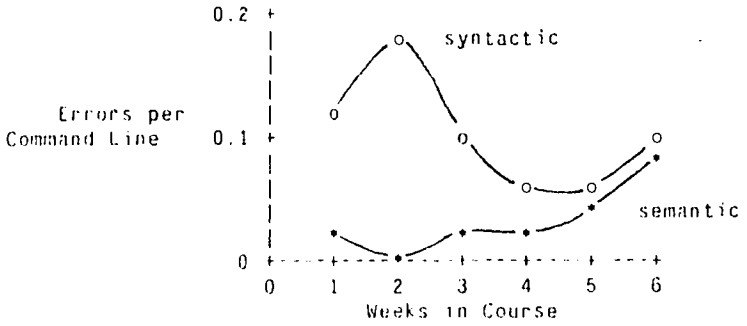


Fig. 11. A Simple Quantitative Analysis of Protocols.

reasons outlined earlier, this technique was employed extensively and uniformly only in the second experiment.

Implicit testing was thus an important part of the curricula and tutoring (e.g. pages 49L & 49S, Appendix 3), apart from the aptitude testing done before both experiments and after the second. It allows students and their mastery of the concepts to be compared at various stages.

The second experiment's post-testing was done only with those students who completed all curricula and projects for both Logo and Simper. The posttest contained questions like those used in the preliminary test, but also asked questions that required writing Logo and Simper programs (see Appendix 2).

The preliminary aptitude test's results were presented as a rank-ordering of the students (Figure 10) obtained by a "forced-choice" evaluation of their work. Perhaps this is not justifiable, for a test whose validity remains uncertain. At least a few students, especially near the medians, might well be reordered or considered hopelessly tied. Yet rank-ordering enforces transitivity. The theory behind the test is simple and qualitative: take as questions examples of the thinking that programmers are typically asked to do, where some types of thinking are more important, in the programming sense, than others. The former relates to validity, the latter to transitivity. No part of the theory suggests cardination or interval scaling. Perhaps a careful, subjective evaluation of students' constructive answers can more nearly approximate an objective ranking-technique (if one exists) than falsely objective testing/scoring procedures can. The theory behind the test may be wrong or incomplete, but determining that is one purpose of the experiments: what do students' interactions with the preliminary test have to do with their interactions with the programming curricula? The test's validity teeters on the subjective choice of questions, and stands or falls subject to experimental data.

5 Results

Anecdotal and judgemental information will be presented which helps in: (a) understanding the students, (b) evaluating the tests, programming languages and curricula, and (c) characterizing relevant features of the tutorial process.

Apart from normally recorded data (replay files and tutorial notes), students provided both direct and indirect feedback in both experiments by explicit opinions and by their behavior. Figure 12a summarizes students' responses to a questionnaire they received shortly after the first experiment. The total numbers of opinions for all rows are not identical because some students felt insufficiently exposed to every item to render an opinion. After the second experiment, a somewhat more qualitative questionnaire was given, but only to a few students who had finished both curricula and some project. Their comments appear in Figure 12b.

Most of the feelings expressed in Figure 12 correlate with casual comments made by the students during the experiments. In the first experiment for instance, the plotter was preferred to the robot because "it draws better" (it produced more faithful drawings); the plotter was preferred to Logo graphics because it produced portable, permanent results; and Logo graphics was preferred to the robot because it was faster, more accurate, and personally available for each student. In the second experiment, more emphasis was placed on the languages and concepts, but most students still expressed clear preferences for graphics and Logo over teletypewriters and Simper, despite the addition of full graphics capability to Simper. Graphics instruction in the second experiment occurred only at the end of either curriculum and was related to a project chosen by each student reaching that point. Thus, each student's liking of graphics and animation was a function of his or her feelings about the project(s) chosen. For example, one student chose to implement a graphic

Tone of Student Remarks

Subject	Negative	Noncommittal	Positive
Plotter		1	16
Graphics Turtle		2	26
Games		3	25
Tutors	2	3	25
Return again	2	3	25
Train		3	14
Robot Turtle	1	1	9
Logo		8	21
Logo Lessons	3	8	18
Simper Lessons		5	5
Simper	3	3	5
Teletypewriters	4	12	11

Subjects are ranked on relative fraction of positive remarks.

Fig. 12a. The 1973-Students' Preferences

Likes:

"I got to learn two languages and I was able to better understand the difference between languages machines understand and languages people understand."

"Liked everything about it and had a great time."

"I liked being able to use letters as well as numbers in writing programs-- I was able to write programs using words and sentences, not just numbers."

"It gave me something to do."

"I liked the experience of getting to know them [Simper & Logo]."

"Everything was A.O.K. including the teacher ... always willing to help."

"The amount of time [plenty of it] to do things."

"I like the fact that Logo is so easy to follow."

"[The curricula were] well written, ... and I feel I learned alot. I also think the teacher did a good job."

Dislikes:

"I didn't really learn that much, you would just learn something and then forget it. It either was so easy or I didn't understand it and got boring."

"Simper ... [I can't follow] where it goes next as easily as in Logo."

Suggestions for Improvement:

"There should be a little bit of discussion for everybody before the beginning of each class."

"Have a few review sheets and review 'quizzes'."

"Drop Simper."

Fig. 12b. Some 1974-Students' Opinions

ping-pong game complete with scoring (Appendix 4). In doing so he learned virtually all there was to know about the graphics system and thence rated using graphics first among his experiences.

The item listed as "games" in Figure 12a refers to certain programs accessible to students on the IMSSS system, such as Hangman, which were intentionally not announced until the students completed most of the curricula. Some students, of course, accidentally discovered a game or two. The policy was that games could be used after a student's regular session with Logo or Simper. Features of popular games are mentioned in report-1. Students were encouraged to write their own games and some were used as examples in the curricula, particularly in the second experiment (see Appendix 3).

Since, as outlined earlier in Chapter 3, the tutors generally fell short of expectations in the first experiment, their highly favorable rating in Figure 12a could provide ammunition for those who believe that students are incapable of appraising their teachers on educationally relevant grounds. The remarks in Figure 12b, however, evidence some astute thinking; particularly the first and third, which are beyond expectations. The student who felt she hadn't learned much also wanted quizzes and reviews, she was apparently not aware of the testing implicit in the curricula and needed clearer motivation. Her faint praise that: "It gave me something to do", also points to a lack of motivation. Furthermore, she had done some programming in Basic in school and never truly saw the value of Logo's more general structure. Unfortunately, she enrolled late and her preliminary feelings aren't available for comparison with those of others from her school (Figure 13).

One prevalent opinion among students familiar with both Logo and Simper was that "it's harder to do things in Simper". So most students preferred to work with Logo, regardless of which language they started with. Figure 14

The School-Teacher's Question:

"I wish to be ... in the 8-week session of computer programming, being offered to junior high students with little or no experience with a computer or computer language. Tell what contacts and interests you have that prompt you to want to take advantage of this opportunity and to be involved with a computer and computer programming for an 8-week session."

The Prospective Students' Responses:

"Computers fascinate me and I really would like to learn some of the ways a computer can be used. I have never used a computer before, but I have seen people using computers and programming them."

"I think it would be interesting to learn the computer language. I have used computers before and have enjoyed it very much."

"I like math and figuring out equations and other things like this. I have done some work with computers, but not very much at all, and I haven't done any work with Logo. I would like to have some sort of career dealing with mathematics, and computer programming would be very interesting and fulfilling."

"My dad is a student at the ... school and talks about computers and how they can solve problems. I would like to learn how to use them myself and also be able to talk 'computer language' with my dad."

Fig. 13. Some 1974 Students' Preliminary Feelings

tabulates the proportion of time that students spent using Logo (and, by complementation, spent using Simper). Note that, within each group, students are ordered by pretest rank. Thus Figure 14 may be matched with Figure 10 to obtain further information. This convention will be observed in other figures in this section, whenever it is appropriate.

In the first experiment, few students finished the Logo curriculum, so Group I spent negligible time with Simper (Figure 14a). But many Group II students went far enough with Simper to be able to start Logo, partly motivated by seeing their friends' work. In the second experiment, more time and a somewhat shorter curriculum allowed Group I students to spend some time in Simper (Figure 14b).

In either experiment, Group II's behavior shows that once students began using Logo, they stayed with it, almost excluding further work with Simper. Figure 14 also shows that students using Logo and Simper simultaneously (Group III), subject only to the stricture that Logo and Simper curriculum parts alternated, chose to spend most of their time with Logo (apart from one, Figure 14b third from bottom, who nearly excluded Logo work, spending time on a Simper number-guessing game). Group III answered a capability question: students can learn two languages, nearly simultaneously, and can do so at least as fast as students who learn the same languages sequentially.

Mass preference of Logo to Simper was a desirable outcome in terms of the students' computer literacy. Although Simper provides a convenient way to learn and experiment with assembly/machine-language programming, it was hoped that students would see the advantage of a high-level language. Indeed, Logo offers what many students seem to want: easy access to message and picture processing. It offers a computationally more important feature: ease of phrasing complicated control structures. However, appreciation of this latter idea was usually confined to the more able students.

(Logo hours / Simper + Logo hours, versus pretest rank,
"-" denotes students who took the test but not the course)

```
Group I

 .99   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .98   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .9[illegible]   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx

Group II

 .70   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .34   XXXXXXXXXXXXXXXXX
 .48   XXXXXXXXXXXXXXXXXXXXXXXX
 .22   XXXXXXXXXXX
 .31   XXXXXXXXXXXXXXXx
0.0
 .17   XXXXXXXXx
  -
0.0
 .04   XX
0.0

Group III

 .82   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .69   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .64   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  -
 .87   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .67   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .69   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .68   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .68   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .88   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Fig. 14a. Breakdown of the 1973-Students' Programming Time

(Logo hours / Simper + Logo hours, versus pretest rank)

Group I

```
 .83   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .92   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .84   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .80   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
1.0    XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Group II

```
 .32   XXXXXXXXXXXXXXXX
 .58   XXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .61   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .16   XXXXXXXXX
 .01   x
```

Group III

```
 .86   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .68   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .82   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
 .13   XXXXXXx
 .71   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
 .63   XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXx
```

Fig. 14b. Breakdown of the 1974-Students' Programming Time

Before further discussing the students' behavior, something can be said about the validity of the preliminary test. For Group II, Figures 14a and 14b indicate a strong correlation between students' ranks on the pretest and the time they needed to complete the bulk of the Simper curriculum--Pearson (Kendall) correlations of .9 (.8) and .6 (.4) respectively, the latter reduced from .9 (.7) because the first student could not stay in the experiment long enough. Students were also ranked subjectively according to final programming ability and dedication to the tasks presented to them in the curricula. Figures 15 and 16 show these ratings, again by pretest rank, for all students.

For the first experiment, Figure 15a also tabulates the mean rate of errors in each student's commands throughout his or her work with Simper. Some slight, joint trend of error rate and pretest rank seems evident. However, averaging errors in this way blurs the nature and importance of individual errors. Without referring to detailed protocol analysis, such a correlation merits little more than a "that's nice". For example, typing and reading ability varied greatly among the students. Furthermore, some students forged along, not caring how many errors they made, while others worried inordinately about making mistakes, particularly observed ones. Various combinations of such abilities and attitudes obviously can confuse simple comparisons of error rates. It happens that the fourth-ranked student (Figure 15a, with a high error-rate) fell into the "unbridled typist" category; the third and fourth from the bottom (with low error-rates) were extremely careful, tending to work out commands on paper before typing them; and the fifth from the bottom had a penchant for typing random numerals, which never appeared as errors because Simper was perfectly happy to store them away. Apparently anomalous error-rates often had explanations that bore directly upon correlations of pretest rank and error rate.

Examining the "mastery" and "perseverance" columns of Figure 15a, we also

Groups II and III (Simper data)

("-" denotes students who worked less than 3 hours)

Errors per Command		Rankings Based Upon Subjective Evaluation of Performance: Mastery	Perseverance
.06	XXX	1	3
.14	XXXXXX	3	2
.11	XXXXx	3	3
.26	XXXXXXXXXXXX	4	3
.03	x	2	1
-		-	-
.07	XXx	3	2
.07	XXx	2	1
.16	XXXXXXX	4	1
.31	XXXXXXXXXXXXXXX	5	4
.23	XXXXXXXXXXx	4	4
-	-	-	
.26	XXXXXXXXXXX	5	4
.50	XXXXXXXXXXXXXXXXXXXXXXX	5	5
-	-	-	
.26	XXXXXXXXXXX	4	4
.15	XXXXXXx	6	5
.13	XXXXXXx	6	2
.16	XXXXXXXX	5	4
.34	XXXXXXXXXXXXXXXX	5	4
.27	XXXXXXXXXXXXx	6	3

Fig. 15a. 1973 Simper Students' Performance Versus Pretest Rank

Groups I and III (Logo data)

("-" denotes students who worked less than 3 hours)

Errors per Command		Rankings Based Upon Subjective Evaluation of Performance: Mastery	Perseverance
.16	XXXXXXXX	1	1
.13	XXXXXXx	2	1
.28	XXXXXXXXXXXXXX	2	1
.33	XXXXXXXXXXXXXXXXx	3	2
.32	XXXXXXXXXXXXXXXX	2	2
-		-	-
.35	XXXXXXXXXXXXXXXXXx	4	5
.22	XXXXXXXXXXX	5	5
.26	XXXXXXXXXXXXX	6	5
.21	XXXXXXXXXXx	2	1
.16	XXXXXXXX	4	4
.15	XXXXXXXx	5	3
.24	XXXXXXXXXXXX	4	2
-	-	-	
.26	XXXXXXXXXXXXX	2	1
.26	XXXXXXXXXXXXX	6	4
.19	XXXXXXXXXx	3	2
.28	XXXXXXXXXXXXXX	5	4
.17	XXXXXXXXx	5	3
.29	XXXXXXXXXXXXXXx	3	2
.15	XXXXXXXx	4	2

Fig. 15b. 1973 Logo Students' Performance Versus Pretest Rank

("-" denotes student who worked less than 3 hours)

Errors per Command		Rankings Based Upon Subjective Evaluation of Performance: Mastery	Perseverance
Groups II and III (Simper data)			
.08	XXXX	..	2
.08	XXXX	2	2
.20	XXXXXXXXXX	1	2
.07	XXXx	1	1
.04	XX	1	1
.04	XX	2	3
.04	XX	1	1
.09	XXXXx	2	2
.09	XXXXx	2	3
.05	XXx	3	1
.07	XXXx	3	2
Groups I and III (Logo data)			
.23	XXXXXXXXXXXx	1	1
.12	XXXXXX	1	1
.19	XXXXXXXXXx	2	2
.14	XXXXXXX	1	1
.23	XXXXXXXXXXXx	2	1
.09	XXXXx	1	1
.21	XXXXXXXXXXx	3	3
-		-	-
.19	XXXXXXXXXx	2	1
.20	XXXXXXXXXX	3	2
-		-	-
.20	XXXXXXXXXX	3	3

Fig. 16. 1974 Students' Performance Versus Pretest Rank

see some mutual trends with pretest rank. High rankers, especially in mastery, tend to be above the median; low rankers below. Figure 15b shows similar results for Logo students. Note, however, the lack of obvious mutual trend between error rate and rank in Figure 15b.

Protocols provide the following explanations. In Groups I and III: the unbridled typist returns with a [illegible]nd as the fourth-and fifth-ranked students; careful planners are bottom and [illegible] from the bottom; the random-numeral typer is now caught by Logo, generating a higher rate, sixth from the bottom; and a new phenomenon: picture-printers, fifth, tenth and eleventh from the bottom, who discovered how 'PRINT" commands could be employed in procedures that "drew" their favorite things (like the "Starship Enterprise"). The latter three students made relatively fewer errors because they stagnated at this point in the curriculum. Students were never coerced to continue the curriculum. Rather, a wait-and-see attitude was adopted, hoping that stragglers would eventually notice that other things, being done by other students, could also be interesting. This tack failed with one of these three students from the first experiment.

In the second experiment (Figure 16), there is again little common trend between error-rate and pretest rank. But again, from protocols, notable exceptions can be explained. For instance, the third student with Simper data has a high error-rate because more than half of all his errors were made playfully, in response to a naming error-message he received one day when he tried to save a program under an illegal name. The middle Logo student has a very low rate because he authored several games (notably graphics ping-pong) which he and others used a great deal, and with little chance for error. As in the first experiment, test rank and subjective evaluations are correlated somewhat.

In general, students experimented more with Logo than they did with Simper, apparently because they felt more able to express their ideas in Logo. This partially explains why the median error-rates in the two experiments for Logo students (.24 and .20) are higher than those for Simper students (.16 and .07). An additional cause is simply that one has a wider variety of errors to commit in a Logo command. This had more noticeable effect in the second experiment (Figure 16). The lower overall error-rate of students in the second experiment also correlates with their apparently better performance on the preliminary test (compare proportions above performance breaks in Figures 10a and 10b). More prompt and accurate tutoring also tended to reduce the total of errors.

Understanding the Students. Here the central interest is, of course, the processes through which students learn programming. The goal being to find observations that shed light on student/tutor interactions in general. The following results derive primarily from detailed protocol analysis, and begin with a sampling of the students' initial, unfettered expectations about computers as expressed first to Simper:

HELLO WHAT'S NEW?	DO YOU WANT TO PLAY JOTTO?
DO YOU LIKE SUMMER?	I AM FUNNY
THIS TYPEWRITER IS TOO SLOW	SOME DOGS ARE WHITE
WHAT IS 12X12?	HOW DO YOU WORK?
TEACH ME HOW TO DO A PROGRAM	HOW DO YOU KNOW?
THERE ARE TWO MILLION FLYS IN AMERICA	LET N = G
YOU ARE WEIRD, BUT SMART	MY NAME IS ...
CAN YOU READ AND WRITE?	CAN YOU TALK?
THE MAN IS CROSSING THE STREET	ARE YOU A COMPUTER?
TO BE OR NOT TO BE	PRINT MY NAME

DEAR JUDY, THIS COMPUTER CLASS IS A LOT OF FUN.
EVERY ONCE IN A WHILE THE COMPUTER GOES WACKEY!

then to Logo (some error messages are shown--in lower case):

HOW MANY QUESTIONS CAN YOU ANSWER?

HOW MANY WORDS DO YOU KNOW?

COMPUTERS ARE DUMB
computers needs a meaning
COMPUTERS ARE ILLOGICAL

HOW LONG HAVE YOU BEEN IN SERVICE?
how needs a meaning
YES
yes needs a meaning
AFFIRMATIVE
affirmative needs a meaning
YES MEANS AGREED, CORRECT
yes needs a meaning
I JUST GAVE YOU A MEANING
i needs a meaning
I MEANS #176
i needs a meaning
I GIVE UP

ARE YOU A LOGO OR A COMPUTER?

I HAVE HOMEWORK TODAY,
I HATE HOMEWORK

WHY ARE YOU A COMPUTER?

THIS IS GOING TO BE VERY FUN
this needs a meaning
IT MEANS IT WILL BE ENJOYABLE

MY DOG IS BLACK

THE SUNSET IS BEAUTIFUL

PLAY CHESS
play needs a meaning
PLAY MEANS ' DO SOMETHING

FUN
fun needs a meaning
IT'S LOGO

ADD TWO AND FOUR

GIVE ME AN INTRODUCTION

I AM A VERY BADLY GOOD BOY

I AM IN A VERY GOOD COMPUTER CLASS BECAUSE IT IS A PLEASURE
WORKING WITH THE COMPUTER, SHE, LOGO, RETYPES WHATEVER YOU WANT

Of course, students had been encouraged to plumb Logo's and Simper's "minds", and all the above efforts received replies of no more than either "unknown operation xxx" from Simper or "xxx needs a meaning" from Logo. Interestingly, Logo's more understandable response tended to stimulate dialogs. Some students struck forthrightously upon primitive operations--Simper:

COMPUTERS ARE FUNNY
'are' isn't a register, use a, b, or p
COMMAND YOU
'you' isn't a register, use a, b, or p

('COM' is short for Simper's 'COMPARE' operation), and Logo:

WHERE IS GERMANY?
you are not using the train
YES I AM

RETURN YOUR LIBRARY BOOKS NOW
your needs a meaning

GET GOLF
something missing for get
GET GAME
something missing for get
GET PLAY
something missing for get

```
                                    YOU ARE A STUPID COMPUTER

PRINT "*"
*                                   IS THE COMPUTER A COMPUTER?
                                    the needs a meaning
DO GO GO                            IS GEORGE HOMSY A COMPUTER?
go needs a meaning                  george needs a meaning
                                    IS MR. HOMSY A COMPUTER?
MAKE A SN PY                        mr. needs a meaning
a needs a meaning                   IS HOMSY A COMPUTER?
                                    homsy needs a meaning
BREAK IT UP; YOU NASTY THING!       SHUT YOUR TERMINAL UP AND
break                               GIVE ME AN ANSWER
```

At this early stage, accidental discoveries of this sort usually passed unnoticed. Eventually most students did take notice of and exploited various syntactic features like mindless error-messages, Simper's abbreviation-by-truncation, and the commenting character ';'--Simper:

```
UNKNOWN OPERATION                   WRINKLE A
unknown operation unknown           HALLUCINATIONS

; YOU CAN'T TALK WITH ME BECAUSE YOU ARE DUMB
```

('WRI' or 'HAL' select Simper's 'WRITE' or 'HALT' operations) and Logo:

```
I AM THE TURTLE                     THISCOMPUTER
i needs a meaning                   thiscomputer needs a meaning

PRINT REQUEST
* IF PAUL IS GREAT TYPE THIS SENTENCE OVER WORD FOR WORD

        PRINT "I DO NOT KNOW HOW TO PRINT"
        i do not know how to print
        ; YOU LIED

        IF OREN IS DUMB, TYPE IF NEEDS A MEANING
        oren needs a meaning
```

Paradox and prediction were favorite techniques that some students used to "insult" their friends. To his lasting surprise, the latter above student's trickery hoisted him on his own petard. These were healthy interactions because students were encouraged to keep experimentation as one of their basic learning tools.

Initially, these naive programmers often had a very high opinion of

putational technology. It was easy to show them that English is not yet a mode of communication between human and machine, but it often took a while for the implications of this to penetrate. At times, students' attempts at communication were tied to curriculum ideas:

```
REMARK LITERALLY                PUT A BU   A

SIMPER COMMANDS ARE FAMILIAR TO COMPUTERS LIKE SIMPER

PRINT "HOWCOMELOGOWORDSDONOTHAVESPACES?

WILL YOU WRITE ME SOME SIMPER   ASE

3 4 10 ARE RELATIVE TO THE NUMBERS 15, 17, 29.
IN WHAT WAY THOUGH?
unknown operation 3
3 (THREE) IS A NUMBER AND ALL COMPUTERS LIKE YOU SHOULD
KNOW WHAT IT MEANS!
```

Sometimes they became confused about curriculum instructions for giving commands. The following shows some examples along with the motivating curriculum excerpt:

A FEW WORDS	... type a few words and ...
LINEFEED	... all you do is type LINEFEED and ...
1 TYPING 1	... and then typing 1 and ENTER ...
GO TO THE SUPERMARKET BUY EGGS AND BACON	(see Appendix 3, page 18L or 17S)
FIX PUT P 2 TO P 1 RUN	... use FIX to change ... from PUT P 2 to PUT P 1 and then use RUN and ...

In fact, some students typed Simper's prompt because it had been shown at the beginning of a line they were asked to type:

```
001: ADD A 12
unknown operation 001:
```

One student tried to get a program to run by simulating Simper's runtime message:

```
EXECUTING 1 TO 250
unknown operation executing
```

producing an enjoyably idiotic response. Another student, in his frustration, uncovered a bug; not in one of the interpreters, but in the Sail compiler's string runtime-routines:

```
,YOU STUPID COMPUTER
'stupid' isn't a register use a, b, or p
```

The bug disguised the "," and thus the proper erro . "unknown operation ,you". Once in a while, unrestrained joy of accomplishment in solving a problem would surface, only to be muted by an unbending parser:

```
YAHOO! I DID IT!
unknown operation y   oo!
; I KNOW I KNOW
```

Obviously this student already knew how to protect her comments.

Confusions sometimes arose w n students worked with both Logo and Simper (a the Group III). Logo commands cropped up in Simper protocols and vice-versa. In these cases, however, the first or second error message usually w s ficie to remind the student of which interpreter was listening to his or her typing. In a few cases, students thought they could resort to Logo commands when their Simper programs failed to produce results. This was one simple way students gave evidence of being more at ease with the Logo language. By far the most common interjection of Logo commands into Simper protocols was in saving programs. Apparently, learning the more complicated Logo scheme of "entries" in "files" overrode some students' knowledge of Simper's simpler filing method.

At the very least, most students initially thought that a computer coul help the n on a personal basis:

```
PRINT "ALL THE C URSES AND LESSONS YOU H VE TO OFFER"
```

Agreed; that should, and perhaps will, s meday be the case. Several students

discovered Simper's '?' (or 'HELP') command which printed a general description of the Simper language. While this was never intended to be a necessary part of the course, it nonetheless was exercised frequently by a few students. Curiosity and an open desire for aid were attitudes to be exploited and encouraged. Students' were willing to experiment in trying to use Logo and Simper as information resources to help them work on ideas from the curriculum.

Now, in discussing details of how students learned the concepts and the languages, the Simper and Logo protocol data will be treated separately. Some observations relating students' performance and their work in the preliminary test will also be mentioned.

Simper. Since work with numbers was so much a part of these and prior schooling, it was relatively easy for them to accept that a machine (Simper) could have a good memory for numerals. But several had difficulty understanding that some numerals could have special meaning, other than counting, to a machine. In the first experiment, this was a problem because of the premature introduction of assembly language, thus working downward from English rather than upward from machine language. The latter sequence was adopted in the second experiment and reduced the incidence of syntactic errors such as multiple instructions per line, making it clearer that only three fields can be assembled into one memory cell's machine-language numeral.

The orderly execution of numerals as instructions was still more abstract. The shopping-list example (Appendix 3, page 178) and the house-to-house collection (Appendix 3, page 318) failed to motivate [illegible] execution for some students. Programs were written with interspersed "holes", despite the obviously sequential relationship between instructions on either side of a hole. A self-destructing program used in the first experiment helped here (see report-

1), and in the second experiment, greater care in introducing machine language seemed to be sufficient. Some of the "holey" programming can be traced to Group III students who learned to use Logo line-numbers in canonically sparse (10-20-30...) sequence and hoped the same editing advantages would accrue in Simper.

In both experiments, addressing values rather than stating them directly was difficult for many students. One wrote his own time-telling program, knew what had to be done to get minutes from seconds, knew something about auditing already, but typed:

```
001 :TIME A
002 :DIVIDE A 60
```

though he did not intend to divide by the content of location 60. The curriculum section on indirect addressing was very helpful to those students who still had trouble with this concept. Not surprisingly, students who had trouble with the implicit name-value associations of the numbers-in-boxes problem on the preliminary test also had trouble with addressing in Simper.

The most pervasive problem was mastering the concept of context (or locality of information) both from the student's point of view as a user and from the point of view of instructions within his or her programs. The most common example of the former occurred when a student ran a program and decided that it needed modification. While it was still running, and perhaps waiting for an input (for 'CASK' or 'ASK'), he or she would type an editing command (e.g., 'LIST' or 'SCRATCH'), fully expecting it to be obeyed. This runtime/edit-time confusion was seen in every student's work at least once.

Context errors within programs centered upon redundant or memory-"clobbering" sets of instructions. For instance:

```
001 :PUT B 1                        001 :PUT B 1
002 :STORE B ONE                    002 :STORE B ONE
003 :ASK A              or          003 :ASK B
004 :PUT B 1                        004 :STORE B @A
005 :STORE B ONE                    005 :PUT P .-3
```

In the first program, the contents of register B and cell 'ONE' are unnecessarily reset at 004 and 005; in the second, the content of cell 'ONE' is continually destroyed by 'PUT P .-3' (instead of 'PUT P .-2'). This latter kind of bug was common, yet it had already been exploited as an example within the curriculum for the first experiment. It was apparent that a much more explicit treatment of computational context was needed, and this was done in the second experiment, with mixed results. Students who had the most trouble with the candy-machine problem on the pretest typically had the most trouble organizing their Simper programs.

The most subtle way in which context affected the students was in the relationships among the interpreter, the assembler and the machine. Most students in the first experiment didn't fully grasp the distinction between editing commands and assembler/machine instructions. Sometimes they attempted to abbreviate the former (e.g., "SCR" for 'SCRATCH') and expect the latter to be obeyed at once. The second experiment's curriculum was modified to clarify these issues, which were founded primarily upon the confusion of editing time with execution time. Its better tack of introducing machine language before assembly language helped a great deal and explicit discussions of runtime/edit-time were included. No one question on the preliminary test seemed to relate strongly to this type of error. This is probably one point for improvement of the test.

Toward the end of the curriculum and in student projects, procedures and their calling sequences were provided examples of how programs could be structured by writing functionally related subunits. In this case, holes were ok. Success

here demanded that the student had mastered the concepts of addressing and program control. Failures to structure these programs correctly were of two forms: failure to define a proper calling sequence, and misplacement of the calling sequence in the flow of the program. Some inputs to procedures, particularly the return address, were overlooked; once the call itself was incorporated as part of the procedure body.

Because, in the first experiment, no students had time to do significant work on the final part of the curriculum dealing with stacks and recursive procedures, the second experiment treated these programming techniques only as tools for use in projects chosen by students who had completed the formal curriculum. When these tools were exercised, by a few students, the notion of context could be be motivated very well. However, in either experiment, few students completed the curriculum and fewer still completed some project. In passing through the course, the data gradually becomes dominated by the work of the more able, typically older, students. The remaining students simply did not proceed as far. This has undoubtedly colored later observations.

Before dealing with individual student performance, a few miscellaneous comments remain. Some students actively exploited features of the Simper interpreter--for instance, truncation of operation names (e.g., 'STOP' for 'STORE' and 'LOAN' for 'LOAD'). One student occasionally harassed the machine by repeatedly saving a program on a file that already existed just so he could respond "no" to Simper's warning: "a program called xxx already exists! ok to destroy it?". The importance of clear, relevant error messages also became apparent (see Chapter 4 for examples). An example follows that shows how misreading one word can dangerously alter the meaning of a message.

```
SAVE
what you want to name your program? YES
ok, yes is saved
```

illustrates the care that must be applied to apparently trivial aspects of an interpreter. In line with earlier comments about contextual errors, it should be mentioned that the above question and the students together produced several saved programs called 'SCRATCH'.

Figure 17 displays the sequence in which Simper-related concepts were learned by each of the students in the second experiment, for which the best data exists. The time at which mastery occurred was judged as outlined in Chapter 4, using error analysis. These language-related concepts connect with one or more of the general concepts outlined in Table I, and so give an approximate idea of the sequence of their mastery.

Logo. Students were less able to adjust to Logo's string manipulations than to its more familiar numerical notation. For example, most students had trouble remembering to quote nonnumerical strings. Logo does not require that numerals be quoted, but demands that literal words and sentences be quoted. The former default tended to be generalized by some to their designation of the latter, especially in direct commands. The second experiment attempted to clarify these notational matters, but was not entirely successful--all literals should probably be quoted at first, perhaps even by modifying Logo.

Procedures had not been introduced early enough in the first experiment, so those students did not have a framework within which to execute direct commands and then add them to stored programs by editing. In the second experiment, procedures were introduced early (Appendix 3, page 18L) and as being, in essence, new Logo commands. Many students soon caught on to the value of being able to construct new and personal tools, either for use or amusement:

```
TO SKIPIP :0
5 IF ZEROP :0 THEN DONE
10 SKIP
15 SKIPIP DIFFERENCE :0 1
END
```

```
TO WIDL?
123 P "DIAJ"
124 P "DIAJ NEEDS A MEANING"
125 P "OH, O.K."
END
```

Approximate Hours to Apparent Mastery of a Concept

Concept	Student (by pretest rank, Figure 10b) 1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Addressing	1.1	3.8	1.2	1.3	1.4	1.2	5.1	-	1.3	5	1.7	-	2.3	4.1	2.4	-	-
Successor Execution	.9	1.8	1.3	.9	1	1.2	1.4	1.8	1.6	2.5	.9	-	.9	1.2	1.7	-	-
Simple Control (using 'PUT')	3.2	5.6	8.3	2.6	4.7	3.4	3.8	-	2.8	7.5	3.7	-	4.3	7.7	5.9	-	-
Decisions (using 'JUMP'/ 'COMPARE')	3.9	5.8	8.3	3.6	5.9	3.4	2.3	-	3.3	7.1	3.4	-	5	7.7	5.9	-	-
Iteration	-	-	-	-	8	8.7	7.2	-	7.3	-	-	-	-	-	-	-	-
Sub-Programs	-	6.6	-	-	6.9	8.2	7.3	-	9.1	8.5	7	-	7.8	10.6	-	-	-
Internal Context (using storage)	.9	1.8	6.7	3	1.6	1.2	2.4	-	4	9.3	1.8	-	1.5	9.1	1.7	-	-
User Machine Context ('RUN'-'EDIT')	3.2	6.2	8.6	2.5	5.6	1.2	2.4	-	1.3	7.8	2.4	-	6.4	5	5.3	-	-

(Boldface numbers indicate very accurate times, a dash signifies that a concept was never clearly mastered.)

Fig. 17. Timing of 1974 Students' Mastery of Simper-Related Concepts

```
TO TRY
10 BELLS 30
20 P "IF ALEX CANNARA IS YOUR INSTRUCTOR DO NOT I REPEAT DO
      N T COPY SOMEBODY ELSE'S PROCEDURE , OR YOU WILL SPEND
      TWO HOURS TRYING TO FIGURE IT OUT AND ALSO YOU WILL
      MAKE ME MAD!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!"
END
```

The first above was constructed by a student when, well into the curriculum, he needed a way of clearing his display screen and didn't know that Logo had such a command ('CLEAR'). He obviously had already mastered iteration and simple recursion. Interestingly, simple (tail-type) recursion came easily to many students upon their first exposure (Appendix 3, page 20L), and common ad-libs took the form:

```
TO RING                 TO BICYCLE
10 BELL                 10 P "LIBBY SHOOP"
20 RING                 20 BICYCLE
END                     END
```

```
TO TOM
10 PRINT "IF TOM WAS NOT GREAT I WOULD STOP WRITING"
20 TOM
END
```

Several procedures (e.g., 'TELLTIME') that were used in the second experiment were incorporated because students in the first experiment had done them on their own and thus found them interesting or useful. Sometimes early procedure-writing attempted the impossible:

```
TO TELLWEATHER
10 PRINT WEATHER
END
```

In the first experiment, naming (name/value associations) had been introduced via Logo's 'MAKE' operation, but there was relatively little use for this in later programming. In the second experiment, procedures were used to introduce the concept, as named chunks of commands which might even receive messages or values (inputs) and link them with internal names (Appendix 3, Logo Part 4). 'MAKE' was never mentioned unless a student's project logically required it.

Typical errors using 'MAKE' in the first experiment were: (1) forgetting quotes around name and/or value, (2) using colons around the name (reasonable in view of most of Logo's syntax, e.g., 'MAKE :X: "Y"' is really not linking the value 'Y' to the name 'X', but to the value already associated with 'X'), (3) inverting name and value positions ('MAKE SUM OF 5 AND 9 ANSWER'), and (4) linking assignments by one command (a reasonable expectation, e.g., 'MAKE "SNOOPY" "CHARLIE BROWN" "LINUS"', where the curriculum intended MAKE "SNOOPY" "CHARLIE BROWN"' and 'MAKE "CHARLIE BROWN' "LINUS"').

Naming errors made by students in the second experiment, where procedures introduced the concept, were reflected by defective input correspondences and control problems generated during editing. For example: (1) input-variable names in the title could not match those in the procedure's body, and (2) the name of a procedure would be edited but not then changed in a recursive call or a call in another procedure.

In both experiments, initial confusions about Logo's colon notation (i.e., ':X:' means "value associated with name 'X'") produced errors like: "PRINT ::SNOOPY::" (to achieve indirect addressing), and: 'RETURN PRODUCT :X: :2:' or 'DOUBLE :324:' (confusions between literals and names, and between actual and formal parameters). Part of the confusion arose because Logo does allow indirect addressing via repeated applications of 'VALUE' ('THING'), and it allows numerals to be names.

Generally, students who had trouble with the candy-machine and numbers-in-boxes problems on the preliminary test also had trouble with procedure formation. The concept of variable enters at several points in producing a working Logo procedure and, just as in Simper, is not confined to the runtime/editing rhetoric. One must also consider the concept of variables

formed in multiple or recursive procedure calls or in complicated command lines. The linkage is more subtle in Logo since it is managed by the parsing/execution stack (see Chapter 2, or report-1), and most students' misunderstandings showed up as soon as they tried to solve problems requiring more than one procedure, or even more than one input to one procedure:

```
TO FUNNYADD :SOMETHING: :SOMETHING:
10 RETURN SUM FIRST :SOMETHING: FIRST :SOMETHING:
END
```

The above, when executed, e.g., by 'FUNNYADD 87 15', will not return 9 but 2 instead, because only the last instantiation of ':SOMETHING:' will be on the execution stack when line 10 is executed. This student simply thought that the position of a name in a title line, rather than its character content, linked it to a command-line input.

The 'DOUBLE' procedure, given as an example which students were to later modify (Appendix 3, page 28L), provides an exemplary set of errors made by students early in either experiment. Only the command line and not the title are shown here:

```
10 RETURN MULTIPLY :X: 2        10 RETURN PRODUCT :X: :X:
```

The first is a linguistic confusion: should an operation's name reflect its result (product) or its action (multiply)? The second is a very common error that unintentionally makes a squarer--the squaring operation itself being unknown to most students who made this error!

Since Logo accepts "noise" words such as 'OF' and 'AND' (e.g., 'SUM OF 2 AND 3'), many students expected to be able to use "BY" or "TIMES" in appropriate places in 'DOUBLE' or its inverse: 'UNDOUBLE'. The pros and cons of noise words will be discussed later. Examples of personal noise words and other errors made by students doing 'UNDOUBLE' follow:

UNDOUBLE MEANS TO DIVID	TO UNDOUBLE IS TO TAKE HALF
RETURN DIV 2 :NUMBER:	TO UNDOUBLE :THING OF :NUMBER:
RETURN QUO :NUMBER: :2:	PRINT DIVIDE :NUMBER: BY 2
RETURN QUO :NUBER: :NUMBER:	PRINT DIVISION :NUMBER: :NUMBER:
RETURN QUOTIENT :NUMBER: BY 2	PRINT QUOTIENT :NUMBER: DIVIDED BY 2

RETURN QUO OF :NUMBER: AND :NUMBER: BY 2

Some classes of error already discussed appear here, namely English attempts at solutions, spontaneous noise words, and name/value errors. An additional problem is evident that concerns the stream of messages processed by Logo during command execution, namely: to print or return a computed value. Many students seemed to think that the printing on their terminal was examined by Logo at the same level as a command. One student believed she needed to comment (with ';') part of a string because only its first word was a legal Logo operation:

```
TO BY
10 PRINT "GOODBYE; KAREN. SEE YOU TOMORROW!!"
20 GOODBYE
END
```

Thus students had trouble understanding that the receiver of a message determines its context and thus its meaning (or effect). Some were particularly confused and thought that they must, for example, say: 'PRINT UNDOUBLE 3' even if their 'UNDOUBLE' properly contained a 'PRINT'.

The contrast between 'PRINT' (or 'TYPE') and 'RETURN' was also based upon the execution-control aspect of 'RETURN'--it terminates a procedure when executed, no matter where it appears. This was typically a problem for some students, who used multiple 'RETURN's as if they were appending to the output message, as 'PRINT' does. Typically the several procedures given in the curriculum as exercises (Appendix 3, Part 4) had all to be done before a

student really seemed to master the basic difference between 'PRINT' and 'RETURN'.

In the first experiment, a problem based upon a preliminary-test question (the 2-column function-table, Appendix 2, page 128) was presented in both curricula. Since its command line involved one of the earliest exposures of students to composition of functions, some attempted solutions are interesting. It was hoped that students would use their 'DOUBLE' procedure in the solution:

```
TO RULE :NUMBER:
10 RETURN SUM 9 AND DOUBLE :NUMBER:
END
```

But those not using 'DOUBLE' often became entangled in the mysteries of nested expressions, noise words and syntax in trying to produce: 'RETURN SUM :NUMBER: AND SUM OF :NUMBER: AND 9'. Some examples:

```
RETURN SUM :NUMBER: :NUMBER: 9

RETURN SUM :NUMBER: :NUMBER: SUM OF 9

SUM OF 9 TO THE PRODUCT OF :NUM: BY 2

TO CORRESPOND 3 TO 15, 4 TO 17, AND 10 TO 29

10 MULTIPLY :NUM: BY 2
20 ADD 9

10 MAKE PROD :NUMBER: AND 2 ANSWER
20 RETURN SUM OF ANSWER AND 9

TO ADD :NUMBER:
10 RETURN SUM DOUBLE ADD 9
```

The last example loops forever as 'ADD' calls itself with 9. In the preceding two examples, students appeared to understand the rule but tried writing the expression on sequential command lines, among other errors. Such attempts to communicate values implicitly across command boundaries were initially quite common and not related to prior work with Simper. In some cases, the curriculum (Appendix 3, page 18L) was one influence, but most of these

students simply felt it was a natural way to proceed towards a solution. Again, misunderstanding of context usually was the culprit.

Students were always encouraged to decompose a program into a basic set of related procedures. This was true for graphics projects as well (see report-1). One problem ('SWITCH13', Appendix 3, page 37L) was quite effective in demonstrating this principle, particularly in the second experiment because of the earlier introduction of procedures. Errors in solving this problem and other, like problems involved coordinating procedure inputs, choosing operations, and use of the 'RETURN' command. Students who forgot to declare input names in the title, or used names different from those named in the title, found that Logo happily supplies them with the default value '""' rather than complain about an undefined variable. A desirable solution was:

```
TO SWITCH13 :X:
10 RETURN WORD THIRD :X: WORD SECOND :X: WORD FIRST :X:
    BUTFIRST BUTFIRST BUTFIRST :X:
END
```

where 'SECOND' and 'THIRD' were previously written by the students to return the second and third letters of a word respectively. Students often failed to break the problem into manageable parts and thereby notice that some of the components had been solved previously. An acceptable solution of that ilk was:

```
10 RETURN W F BF BF :X: W F BF :X: W F :X: BF BF BF :X:
```

('BF' abbreviates 'BUTFIRST'; 'F', 'FIRST'; and 'W', 'WORD'). Actual attempts:

```
10 RETURN W W W F BF BF F BF F BF BF BF :W:
```

```
TO SWITCH13
10 THIRD :INPUT:
20 FIRST :INPUT:
30 PUT THIRD FIRST AND FIRST THIRD
END
```

The first example shows a common initial belief that one input can be

distributed over several operations. The second shows attempted inter-line communication, implicit 'RETURN' and English instructions. A related, simpler procedure, to put the first letter in a word last, was written by one student as:

```
TO REV :YIP:
10 RETURN :IPY:
END
```

in the interesting belief that characters in an input's name map into those of its value. Because Logo defaults undeclared names, as mentioned earlier, she persisted with this scheme in several procedures, thinking she only had to get the right combination of letters to succeed.

One frequent error was forgetting to specify all of the inputs in a direct command or recursive call, especially when that input does not change. One 1973-graphics student defined the following unusual program:

```
TO STEVE :BD 17 16 48:
10 :BD 17 16 48:
END
```

```
TO BD :L: :A: :I:
10 FRONT :L:
20 RIGHT :A:
30 BD :L: SUM :A: :I: :I:
END
```

She then typed 'STEVE BD 17 16 48', which works (in the sense that 'BD' is executed), because in attempting to bind the input, Logo runs 'BD' and waits for a value, which never comes. The student did not seem to realize this, trying 'STEVE' with a different call to 'BD', with 'STEVE' and 'BD' traced, would have helped to correct this mistake.

Many students had trouble understanding how procedures communicate values to one another via 'RETURN'. In the second experiment, for example, students wrote many procedures that were to return values:

```
TO COMPARE :SOME: :TOY:
10 TEST NUMBERP :SOME: :TOY:
20 IFFALSE RETURN "FALSE"
30 IFTRUE EQUALP :SOME: :TOY:
END
```

When constants (e.g., '"FALSE"') were to be returned, 'RETURN' was rarely forgotten, but when another operation/procedure was to be called to generate the returned value, 'RETURN' was often forgotten or assumed to be implicit, as in line 30 above. Several students used structures like line 30 to mean: "now be 'EQUALP' and do what it does"--an attempt to implicitly change a procedure's definition at runtime.

Some projects (e.g., 'BINAR', report-1 or Appendix 2, page 127) were taken from the first experiment's curricula and used as part of a posttest for students who completed all of the second experiment's curricula. Other projects were used for the implicit testing process outlined in the tutoring discussions earlier, and most students added their own, especially when they were able to use the graphics system. Some are mentioned in report-1. In the second experiment, for instance, one student designed a simulation of the PONG(R) game and another began an animated cookbook that was supposed to implement a recipe visually by allowing the user to manipulate snapshots of spoons, cups, etc. Some of these projects are documented in Appendix 4.

As done earlier for Simper (Figure 17), Figure 18 displays the apparent sequence in which Logo-related concepts were learned by each of the students in the second experiment, for which the best data exists. A few students' work will be discussed in detail after some remarks about the languages and curricula.

Approximate Hours to Apparent Mastery of Concept

Concept	Student (by pretest rank, Figure 10b)																
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
Literal Values	1.8	1.7	1.1	1.2	1	1.4	1.7	4.2	.4	-	.5	1.9	.4	1	.5	.6	3.3
Named Values	4.2	5.6	5.6	2.3	3	2	4.6	10	4	-	2	-	-	6.1	3.1	-	4.4
Command Parity	4.2	2.3	5.6	4.5	3.2	2.8	4.6	11.4	4.9	-	6.4	-	-	7.1	7.5	-	6.8
Simple Control (sub-procedure)	2.7	2.3	1.7	1.3	1.5	1.4	2.8	4.1	4.4	-	2.6	-	1.4	1.8	10.2	-	6.8
Simple 'RETURN'	4.2	6.5	9.4	7.8	4.8	2.9	5.5	11.4	4	-	7.8	-	-	6.4	9.1	-	6.8
Recursive 'RETURN' (context)	15.4	18.9	15.5	-	-	17.6	7.9	-	-	-	16.5	-	-	28.1	-	-	-
Decisions ('IF'/ 'TEST')	7.5	10.1	9.4	7.8	-	8.5	12.4	26.2	-	-	9.5	-	-	10.7	-	-	6.8
Stopping Rules (iteration)	11.6	18.1	17.1	11.4	-	15.2	7.9	-	-	-	11.4	-	-	28.1	-	-	-
User/ Machine Context	3.2	2.3	5.6	1.2	4.8	1.1	2.1	3.6	1.8	-	7.8	-	1.4	2	2.6	-	1.6

(Boldface numbers indicate very accurate times, a dash signifies that a concept was never clearly mastered.)

Fig. 18. Timing of 1974 Students' Mastery of Logo-Related Concepts

Evaluation of Simper and Logo

As a result of the experiments, various modifications were made or should be made to the languages.

Simper. First targets for change have been obvious bugs and inconsistencies in command evaluation and assembly. For example, after the first experiment, 'SCRATCH' was modified to accept the general form for an address-range specification (e.g., 'SCRATCH 6:8' has the obvious effect). 'SAVE' and 'GET" were made to accept the name of the file as an input (e.g., 'SAVE GLOP'), resorting to dialog only when such an input is lacking. A more subtle change was made to 'SLIDE'. One student was frustrated when his memory space was effectively exhausted even though numerous holes existed between program segments. So, by the second experiment, a forward 'SLIDE' (e.g., 'SLIDE 100:200') could recursively squeeze out such holes to make formerly impossible relocations possible. The user is informed of which holes disappear. In the interest of making the name fit the action and to reduce confusions with Logo, 'FIX' was replaced by 'EDIT".

The first experiment also suggested some new operations and a new command. 'LEXOR' gives a decimal version of "exclusive or" (Table II), 'ERROR' tests a flag set by arithmetic overflows, 'IOT' communicates with the Graphics program and the plotter, and 'NEWS' gets the system time schedule and any new information about Simper (or Logo). 'DIVIDE' was modified to set the 'ERROR' flag on division by zero, instead of the previous and unusual skip-if-successful convention. The structure of the Simper machine itself was modified. Five-hundred memory cells and four registers (i.e., A, B, C and P) were made standard (with upper limits as shown in Figure 2). This was motivated by students suggesting projects for which 250 memory cells were insufficient. The additional register was added to make procedure calls more

convenient, especially via a student-programmed stack. The changes were achieved by a generalized restructuring of the interpreter.

After the second experiment, more changes were made, mostly on suggestions of students (see Table III). Significantly, the students were more concerned with improving Simper's editing abili . (e.g., by adding 'FLIP') than with adding new powers to the simulated machine.

Recommendations. Changes are relatively easy to make in Simper because it is written in a high-level language. An important improvement would be the simulation of a micro-coded machine with interrupt handling, so that students could be exposed to some aspects of modern machines. Simulated devices other than the turtle (e.g., a disc) could also be pedagogically beneficial. However, too many "features" can be detrimental. Since a valuable computational idea is that problem solutions can be broken logically into parts that are in turn realized by certain basic and sufficient abilities of some machine, the abilities chosen should not individually be too powerful. A pedagogically useful addition would be the ability to run the machine backwards as well as forwards thus to allow partially undoing a computation.

Perhaps the most beneficial results would be achieved by making the interpreter smarter and more congenial in terms of its responses to naive programmers. A first step would be a structured treatment of the '?' or 'HELP' command. Successive applications of this command in, say, an address field would obtain successively more detailed help about address fields. In this respect, the interpreter would be more knowledgeable about itself. More general (and more difficult) powers, such as the ability to evaluate programs, would be of obvious value in counselling students.

Logo. In the present version of IMSSS Logo (excepting Sailogo), substantial changes are typically difficult to make. For this type of work, the

interpreter should have been written in a high-level language (e.g., Mauis, 1973). Several changes in commands, apart from addition of animation, were made after the experiments (Table IV). Consistency and clarity of nomenclature was the goal. For instance, some Logo predicates mark themselves as such by employing the suffix "P" (e.g., 'LESSP') and some do not (e.g., 'IS'). This was a source of confusion for a few students. 'IS', in particular, is also very suggestive of wrong interpretations (e.g., the line "TEST IS :X: LESSP 0" should be 'TEST LESSP :X: 0'). Thus 'SAMEP' was introduced as an alternative to 'IS'.

Recommendations. Operation names should name the action (e.g., 'ADD') rather than the result (e.g., 'SUM')--or, as the precocious 9-year-old put it: "I'd make a whole new language without any weird commands like 'PRODUCT' and 'REMAINDER'. I'd have MULTIPLY and FINDREMAINDER.". Predicates, rather than simply being suffixed with "P" should end/start with "?" (e.g., 'LESS?'). If ':X:' is to be analogous to 'VALUE "X"', then nesting of colons should be allowed. Additionally, a different symbol should be used instead of colon to delimit place holders in procedure titles, or a different, nestable synonym for 'VALUE' could be chosen (e.g., "@"). Numerals should be disallowed as names or always be quoted when used as literals just as text is. More fundamentally, value names and procedure names should use the same dictionary and notation (e.g., 'A' could either stand for 'VALUE "A"' or call procedure 'A', as in Algol 60). Pedagogically speaking, any distinctions of program from data should be defined by the student and not be automatic and pronomial notation seems most natural.

Another fundamental point concerns command evaluation. Commands for editing, erasing, listing and filing currently quote rather than evaluate their inputs (i.e., 'EDIT ROCKET' instead of 'EDIT "ROCKET"' thus disallowing 'EDIT :R:' where 'VALUE "R"' is "ROCKET"). A consistent, flexible scheme

(assuming names and procedures share the same dictionary as suggested above) would allow only 'EDIT "ROCKET"' and 'EDIT R'. 'EDIT ROCKET' could also be allowed if the user could make his own procedure definitions that quote or evaluate inputs at will--all in the interest of consistency, which is very important to naive programmers. A further simplification would result if one operation (e.g., 'DEFINE' or 'HOWTO') performed the functions of both 'EDIT' and 'TO', since the only difference is the pre-existence of, or lack of, a definition.

Noise words (e.g., 'OF' and 'AND' as in 'SUM OF 3 AND 5') should be eliminated unless they are under user control. 'AND', for instance, has a very strong meaning, almost equivalent to 'WORD', in many students' minds:

```
P SUM OF 3 AND 4 AND 5 AND 6
```

Logo should emulate Lisp in returning values for all commands and perhaps printing these values at the top level rather than giving the message "THERE IS NO COMMAND FOR..." when a student forgets to precede a function call with a receiver for its reply. A user-controlled toggle for automatic value printing would be a useful debugging aid. This would make 'STOP' and 'DONE' equivalent to 'RETURN ""', perhaps leading to their welcomed demise since 'EXIT' really does what their names suggest they do. Error messages should be informative (e.g., "X IS ALREADY A LOGO OPERATION" not "X CAN'T BE A PROCEDURE NAME"). Misleading error messages such as "OUTPUT CAN'T BE USED AS AN INPUT IT DOES NOT OUTPUT" or "OUTPUT CAN ONLY BE USED IN A PROCEDURE" should be avoided (the former is gibberish, the latter should say something like "OUTPUT MUST BE PRECEDED BY A LINE NUMBER"). Error messages should not end with a "?" unless the interpreter is prepared to engage the student in a helpful dialog.

Editing and Filing. At one time or another, most students forget to enter

editing mode with 'EDIT' or 'TO' when trying to change a line in a procedure--commands such as 20 and 'EDL 20' typed at Logo's top level resulted in the messages "LINE 20 OF WHAT' PROCEDURE?" and "EDIT WHAT? YOU ARE NOT DEFINING ANYTHING" which may have misled students into trying the following commands:

```
EDIT LINE 10 OF UNDOUBLE        IN TRI2
ERASE LINE 6 IN RECTANGLE       TO 035 OF RECTANGLE
```

Students often included extra words (some of which Logo had used in its own messages) with operations such as 'EDIT' and 'LIST', which do not obey the general Logo evaluation scheme; hence, error messages were puzzling.

```
EDIT TO EVENP
you can't edit that.

ERASE :XI:
erase what?

END
again defined
UNDEFINE AGAIN
undefine needs a meaning.

LC OF FILE OF MARTA
of can't be a file name
```

```
EDIT :XI:
you can't edit that

ERASE TO SQUARE
erase what?

LIST ALL FILES
list all what?

LIST NAMES
something missing for list.

LIST ALL THAT WAS DONE TODAY
list all what?

GET FILE PC136 VOWELP
file can't be a file name.
```

As a convenience, it might be helpful to allow some default applications of operations like 'LIST'. For instance, when 'LIST', 'EDIT', 'ERASE' or 'EDIT LINE xx' is typed with no input, the default input would be the name of the last procedure defined or executed. Similarly, a one-entry file could be gotten without naming the entry.

The distinction between what is in Logo's immediate memory (workspace) and what is on secondary storage (file entries) seems to be confusing even to adults. By saving an entire workspace on an "entry", it is fairly easy to 'GET'

everything back at a later time. But since the workspace could contain the appended results of several 'GET's from other entries (from other people's files too), there is often unnecessary duplication in 'SAVE's. One should have the ability to save partial workspaces (groups of procedures) on entries:

```
SAVE LIZ D AND UD AND SQUARE    (Liz wanted to save individual
                                 procedures on separate entries)
```

Student typing, some almost verbatim from the curriculum, occurred that one might expect a reasonable computer-based tutor to handle. Merely automating a programming curriculum by typing text at the student accomplishes little in dealing with such questions. Ideally a language interpreter should "know" about concepts and problems the curriculum is presenting and the intents of procedures the student is writing:

```
HOW MANY INPUTS DOES "MAKE" HAVE?

IS REQUEST A LITERAL?
literal needs a meaning.
NO IT DOESN'T

HOW MANY INPUTS DOES PRINT HAVE

IS "GEORGE" A WORD?
```

The ability to answer these questions is easily given to Logo because the subject terminology (perhaps excepting "literal") is Logo's.

Debugging. Since Logo checks procedure lines for matching quotes and colons at the time they are typed, it would also seem advantageous to report other kinds of syntax errors at "define-time" rather than at "run-time". For example, erroneous numbers of inputs for primitive commands or procedures, and undeclared procedures or names (not defined globally or in the procedure's title) could be reported before exiting editing mode, or upon request. The student could act on these suggestions, editing further, or execute the partially defined procedure while still in editing mode, or exit to work on something

else. This could at least help reduce the amount of time students spend in discovering and correcting syntax errors one at a time. The idea of 'TRACE' should be expanded to allow display of command-line execution, since the pursuing of typically complicated commands rivals the complexity of recursive procedure calls. An ability to undo the last command would also be very helpful, as it is to LISP users.

Implications for Curriculum Design

In the first experiment, reports of tutors about student involvement in different parts of the curricula and their own projects, real or planned, led to changes in the presentation order of the concepts and in the techniques for explaining certain concepts.

For Simper, most changes made for the second experiment centered upon better motivations for: context, sequential execution, addressing and assembly language. The machine's language of numerals would be taught before assembler syntax so that students would grasp the latter's reason for existence as well as its structure. The fact that different languages are appropriate for different interactions with Simper was exploited in discussing computational context. The intercommunication of instructions (e.g., via the registers) within programs was also treated in terms of context. For Logo, the first experiment demonstrated that procedures should be introduced early so students can create useful or enjoyable tools right away.

So, for the second experiment, names were introduced first when naming procedures and again when naming their inputs. This definitely improved student interest. Decision making was also introduced earlier in the second experiment, in both curricula. Students could embark earlier on their own projects, like games, some of which were used in parallel in Simper and Logo. Early work with decision making helped the students in the second experiment

do better when the time came to combine it with other concepts needed, for instance, in general recursion with stop rules.

In both experiments, the curriculum format (see Appendix 3) of path pointers, questions, problems and things to try was generally well-received by students. Certain connecting ideas or processes, such as how expressions are evaluated and how program execution proceeds, are difficult to sequence on paper. The flowchart-like diagrams with boxes and arrows (e.g., Appendix 3, page 38L) were not particularly effective. The younger children had special difficulty with these artifices, for the same reasons they had trouble with the candy-machine problem on the preliminary test. Good yet static representations of essentially dynamic processes are hard to come by. For Logo, the "brothers" with knowledge clouds did test understanding when some of their states were left blank, but were of little help in mapping this understanding into a procedure. Good illustrations of effective metaphors are very important.

One of the questions addressed by this work has been "what are effective metaphors for teaching the concepts (Table I) to naive programmers." For many students, the concept of a context or computational environment proved most difficult. In simplest form this reared itself in their confusing editing and execution times/languages when interacting with the Logo or Simper interpreters. Fresh students often gave editing commands to their running programs, not realizing that their programs had, in effect, taken over the machine and defined new languages. A linguist would probably say this is a common problem in human languages as well. The most successful metaphor used in this work involves thinking about the ability of an active entity (machine/animal) to give its attention to some source (internal/external) of messages and process these messages according to some rules (language). Everyone knows what "giving attention" means to himself or herself. Linking this to generalizations about machines (candy/computing) is all that's needed.

This applies directly to explaining functions too, if they are thought of as translators.

In Logo, dealing with recursive procedures that return values was difficult for almost all students. The above metaphor coupled with an analogy drawn to a chain telephone call seemed most helpful. The complication is that each caller must wait (on "hold" or to be called back) until the "callee" has an answer to give. The success fo this tutoring device raised hopefully clearer alternatives to 'RETURN' such as 'REPLY' in Logo. The way in which Logo uses its internal pushdown stack for saving local contexts during recursion (or the equivalent Simper programming) links straightforwardly with the attention metaphor above.

Play-acting out programs, particulary Logo, was tried in both experiments (e.g., Appendix 3, page 39L) with mixed results. It seemed most effective when used to explore command evaluation, coupled with the telephone metaphor and a wary likening of inputs to be instantiated to mailboxes in need of letters. For simple syntactic problems (e.g., how many 'WORD's to use) a little applied logic often produced helpful analogies (e.g., for n values use n-1 'WORD's because it takes n-1 dabs of glue to stick together n blocks).

Case Studies

The problem encountered by two second-experiment students at each point in each curriculum will be discussed in order to expose both their differences and commonalities in thought when faced with the task of learning their first programming languages. They will be referred to by their rank position on the pretest (Figure 10b) and discussed separately for two languages.

Simper. Student 6 worked relatively seriously and, from the start, carefully and thoroughly followed the curriculum instructions and examples.

She was not uncomfortable with the primitive nature of the Simper machine language nor with the basic commands involved in editing. Addressing and successor executions seemed common-sensical to her. After some brief problems with programs that ran off their ends or jumped to nonexistant instructions, she had no further trouble with program control. Her first major project was the number-guessing game from the curriculum. She often did much of the work at home, bringing it the next day to try out. She made two important errors First, picking the number to be guessed but failing to store it in memory for later comparison with the user's guess. Second, using the wrong register in her decision-making instruction. Both errors can be thought of in terms of misappropriating the internal context of the machine at runtime. She also needed help in deciding that the program should pick the number before the user guesses. This would not matter if only one guess were to be allowed. Once her program was working, she used it a great deal and modified both the size of the numbers selected and the hints given when a guess was wrong.

Student 6 went on to other work, but had saved her guessing-game and often recalled it to use. She worked on indirect addressing with no problems. The next important project involved the concept of a data-structure consisting of 5 characters stored in one memory cell. Again she had little difficult and spent time at home working on her program. When the curriculum called for a stop rule to be added to the program for printing 5 characters from a cell, she picked the correct rule with no help. She still had some trouble matching registers correctly in what amounted to a several-instruction program. She also generated a control error by jumping too far back in her program on each loop circuit and re-initializing a memory cell used for counting. This class of error persisted in her work for a few days. A further error in clearing memory at the wrong time prevented her stopping rule from functioning and her program ran on and on. After correcting these errors, it was obvious that her program

almost worked but stored only a partial result in memory--the full result being in a register. She noticed this with no help and corrected the problem. A subsequent attempt at a similar program demonstrated that complex control was still not mastered--a jump was redundantly included and a target symbol was placed one instruction late in the program. In addition, context problems with assumed register content recurred. The program was eventually corrected with help. She then went on to use the graphics system and constructed several iterative drawing programs without error. She then began the Logo curriculum.

Student 14 began Simper and had difficulty immediately in understanding successor execution. His reaction to an erroneous program was to erase it rather than edit it. He was mystified by the first program in the curriculum because, when run, it gave no visible result until memory was displayed after execution. This resulted from a misreading of the curriculum instructions. As a result, he required more than average amounts of tutoring. He had great difficulty understanding the need to match register names when communicating values among instructions. When introduced to addressing, he attempted to address a value by content. That is, he used an address equal to the value, not an arbitrary address, as desired. In doing so, his first such program generated overflows by dividing by empty cells (0 values). He was helped to correct these problems and still preferred to erase entire programs rather than edit. In working on one program from the curriculum, he demonstrated a typical context error: typing editing commands to a running program. At this time he decided to review the entire curriculum. He repeated some previous errors, in particular, use of the wrong registers for inter-instruction communication. Since he was in Group III, he was also learning Logo and some Logo editing commands crept into his Simper interactions. His review of addressing helped him clear up his old confusion about content versus location. In reconstructing one curriculum program, he produced a control structure that jumped to a

wrong location. He also neglected a printing instruction even though he created a value in a register to be printed. Context errors from typing editing commands to running programs persisted.

He made several syntax errors that indicate he doesn't really understand the 3-field structure of Simper instructions. Most notable was an attempt to use multiple address fields to store multiple characters in a register. Again a control problem appeared as he started the guessing-game project. His program had a jump to a redundant instruction. His corrected program worked but printed out a message backwards. When translating a 'JUMP' to a 'COMPARE' he left in an unnecessary instruction prior to the jump that prevented the program from operating. He repeated this error twice. Upon first exposure to symbols, he forgot to attach them to memory cells and so generated illegal addresses upon running the programs. In a later program to realize the function 2x+9, he used an address literally and so wrongly operated on an instruction in the program body. The data-structure program he produced contained several bugs including a misaimed jump and a redundant instruction that is never executed. With help he tried several times, but never quite understood how the program was to function. Iteration and symbolic addressing remained unmastered.

Logo. Student 6 had begun Logo after leaving Simper. She grasped intraprocedure control quickly but failed initially at using nested (sub)proceudres. Once helped she went on to create her own version of a simple recursive procedure to print her name. She was initially confused about procedures that return values and what to do with the value. She had no trouble with simple command syntax, but did have trouble with the colon notation denoting named values--she either neglected the colons in the command line or put them around constants. After a few such errors she seemed to understand name/value associations in Logo. A series of several procedures are

faced in the curriculum which demand successively more complex command lines. She used 'LAST" to mean "place this character last" not "take the last character". Commands that build strings out of parts tended to get too few 'WORD's. She used the same name for both inputs of a 2-input procedure thus getting only the final instantiation when it is called. She considered 'FIRST' to act destructively on its input. She failed to use a building block sub-procedure at an opportune time. After several string manipulating procedures she mastered the command syntax, but did not quite know when to use 'RETURN' appropriately. She failed to use a recursive call when it was of obvious necessity. She used 'RETURN's successively, as if they append to an output message rather than terminate execution. She worked on the first major project--the guessing-game and needed help understanding 'BOTH'. In more complicated projects like 'REVERSE' she demonstrated understanding of inputs and control but not quite of recursive 'RETURN's which she tended to leave dangling so that values were fed to Logo not the calling procedure. The use of stop rules was no problem for her.

Student 14 had been learning Simper at the same time as Logo. He began by typing literally from the curriculum (e.g., "CONTROL-N"). He retyped procedures rather than use edit. He attempted to elicit information from Logo by having it print sentences which, of course, have no meaning to Logo. He tended to use previous procedures' structures as solutions to new problems. He had trouble matching input names to procedure command lines. Prefix notation seemed no problem to him, but he did have trouble providing enough inputs to operations in command lines. He also forgot basic syntax items like line numbers and colons. The major project of the guessing game failed on first try because it tested a constant rather than a computed value. At this point he helped another sutdent with earlier work. After much help he had a working guessing game which he used alot. Some Simper commands appear. In the

more complicated recursive procedures he neglected not only recursive returns but stop rules. He tended to misplace stop rules so they never got executed. Once they were working he enjoyed observing such procedures operate on long inputs under 'TRACE'. When the opportunity arose to use an already existing procedure as a tool in solving another problem, he rarely capitalized on it. He began using the graphics system and experimented with various kinds of pictures drawn from building block procedures he'd been given, but produced little original work.

Summary

The two students, whose work has been outlined, suggest the range of abilities that were present during both experiments. Some students took to the curricula and languages quickly and easily, while others did not. As has been discussed, and as Figures 17 and 18 also suggest, the preliminary test seems to order students approximately on ability to complete the curricula. It also seems, from subjective evaluations of the student , to order them approximately on mastery. The more important question of how students learn the concepts is only answerable from case-study data.

The metaphors outlined earlier seem to work because they help students identify with the process they are trying to understand. The two most common, virtually universal misunderstandings of all the students were: (1) misunderstandings of linguistic/computational context, and (2) ill-defined intents. The former applying to both the storage/passing of information within their programs and their interactions with the interpreters. The latter, or fuzzy program specifications, amounts to wishful thinking, wherein the particular interpreter was expected to read the student's mind and run correctly even though, for instance, a command had been left out. Leaving out recursive 'RETURN's, as mentioned earlier, is a typical example in which the student

expects the computer to be the command whose value is not returned. A brief categorization of all errors appears in Table XI.

In terms of the concepts originally selected as important to learning programming (Table I), a somewhat different ordering on difficulty for each student is implied by individual case-studies, at least in the second experiment whose data are best. Typically, however, individual orderings approximate the sequence listed in Table I, with the notable exceptions of: concept 1, due to user/machine context errors, falls at about position 5; concepts 14, 15, and 17, because of internal program-management errors and common difficulty in starting on a reasonable program design, fall last; and concepts 5 and 9 lump together at position 9.

With regard to programming languages and their influence on students, the data strongly suggest that languages should be syntactically consistent, and powerful in both editing and execution capabilities. As one student said after her first hour with Logo: "If computers can understand languages like Logo, can't they understand English?"

Table XI

Categorization of Observed Student Errors and Misconceptions

Use of Language Syntax

- Predicates difficult to master, especially combinations such as 'BOTH'/'EITHER'.
- Making up nonexistent noise words analogous to Logo's.
- Misunderstanding deferred-command parsing in Logo--inputs are read backwards.
- Using infix and postfix rather than the Logo prefix.
- Trying to use ditto marks to copy parts of a line to next line.
- Existence of "holes" in Simper programs.
- Literal interpretation of Simper address field.
- Thinking that changing (Simper) target cell's content changes all instruction's address fields that reference that cell.
- Forgetting to put a value in a (Simper) target cell before accessing it.
- Testing the wrong register in Simper loops.

Sequencing

- Not knowing any or the simplest stop condition on an iteration or recursion.
- Confusion between iterative and recursive techniques--input and return values.
- Jumping inappropriately.
- Multiple commands per line.
- Improperly communicating Simper instructions that destroy rather than pass on contents of registers.

Use of Procedures

- Meaning of input values (using colon : in Logo for both constants and variables).
- Thinking procedure names must say what they do in order to work.
- Distributed or forgotten inputs.
- Returns from looping procedures unforeseen.
- Names of inputs not distinct or assumed to computationally relate to a value (e.g., see page 93).
- Names of inputs not the same in title and use.

Returning Values

- Simple recursion and 'EXIT' is easy, but returning value to self is not.
- Procedure becomes, semantically, the value or function to be returned.
- Last procedure called, in series of calls, returns value for the entire series.
- Distinction between Logo 'DONE' and 'RETURN'.

Table XI (continued)

Storage/Memory

Not understanding that a 'SAVE' can destroy a previously filed program.
Understanding memory in Simper as read-copy/write-destroy; and that it is permanent until changed by a program.

Editing Versus Runtime

Problems editing Logo titles.
Hard to think about runtime when editing (thinking that editing actually executes).
Understanding what 'RUN' means for a program--that the machine's linguistic appearance to the user is redefined by the program.

Problem Solving Methods

Surprised that a problem can be solved or that the computer can carry out a certain command.
Failure to generalize previous solutions to present problem.
Inability to break problem solution into program steps to write.
Multiple-line solutions rather than well-structured iteration or function calls.
Failure to see minimal solutions.
Failure to exploit the style of the programming language (such as the possibility for extra inputs to act as counter or method of passing conditional information).

References

Bassala, G. Man and machine (book review). *Science*, 187, 248-250, 24 January, 1975.

Berry, P. Pretending to have (or to be) a computer as a strategy in teaching. *Harvard Educational Review*, 34, 383-401, 1964.

Bitterman, M. The comparative analysis of learning. *Science*, 188, 699-709, 16 May, 1975.

Bloom, B. Thought-processes in lectures and discussions. *The Journal of General Education*, 1953, 7, 160-169, April, 1953.

Bloom, B., & Broder, L. *Problem-solving processes of college students.* Chicago: The University of Chicago Press, 1950.

Bradley, J. *Distribution-free statistical tests.* Englewood Cliffs, N.J.: Prentice-Hall, 1968.

Brainerd, C. The origins of number concepts. *Scientific American*, March, 1973.

Brand, S. *Two cybernetic frontiers.* New York/Berkeley, Calif.: Random House/The Bookworks, 1974.

Bredt, T. *A computer model of information processing in children.* Technical Report No. CS100. Stanford, Calif: Computer Science Department, Stanford University, 1968.

Brown, J., & Burton, R. SOPHIE--A pragmatic use of AI in CAI. *The Proceedings of the ACM National Conference*, San Diego, 1974.

Brown, J., & Rubinstein, R. *Recursive functional programming for students in the humanities and social sciences.* Report No. 27. Irvine, Calif.: Department of Information and Computer Science, U. C. Irvine, 1973.

Buxton, J. (Ed.) *Simulation programming languages.* Amsterdam: North-Holland, 1967.

Campbell, D., & Erlebacher, A. How regression artifacts in quasi-experimental evaluations can make compensatory education look harmful. In J. Helmuth (Ed.), *Disadvantaged child. Vol. 3, Compensatory education: A national debate.* New York: Bruner-Mazel, 1970.

Cannara, A. Toward a human computer language. *Creative Computing*, September-October, 1975.

Cannara, A., & Weyer, S. study of children's programming. *Proceedings of the 1974 Confer on Computer-Based Learning Systems*, University of Hamburg, Feder epublic of Germany, August, 1974.

Cannara, A., & Weyer, S. Programming languages for children. *The Second Annual Computer Science Conference*, Detroit, February, 1974.

Carbonell, J. *Mixed initiative man-computer instruction.* Report No. 1971). Boston: Bolt, Beranek & Newman, 1970.

Chaitin, G. Randomness and mathematical proof. *Scientific American*, May, 1975.

Chapanis, A. Interactive human communication. *Scientific American*, March, 1975.

Dahl, O., Dijkstra, E., & Hoare, C. *Structured Programming.* New York:Academic Press, 1972.

Davis, M. (Ed.) *The undecidable.* Hewlett, New York: Raven Press, 1965.

Davis, M., & Hersh, R. Nonstandard analysis. *Scientific American*, June, 1972.

Drake, S. The role of music in Galileo's experiments. *Scientific American*, June, 1975.

Dwyer, T. A. *An experiment in the regional use of computers by secondary schools.* Final Report, NSF-OCA-GJ1077-SOLO, 1972.

Ellis, A. *The use and misuse of computers in education.* New York: McGraw-Hill, 1972.

Evey, R. J. *The theory and applications of pushdown store machines.* Unpublished doctoral dissertation, Harvard University, 1963.

Feigenbaum, E., & Feldman, J. (Eds.) *Computers and thought.* New York: Mcgraw-Hill, 1963.

Feldman, J. A formal semantics for computer languages and its application in a compiler-compiler. *Communications of the ACM*, January, 1966.

Felix, L. *The modern aspect of mathematics.* New York: Basic Books Inc., 1960.

Fenichel, R., & Weizenbaum, J. *Computers and computation.* San Francisco: W. H. Freeman and Co., 1971.

Fenichel, R., Weizenbaum, J., & Yochelson, J. A program to teach programming. *Communications of the ACM*, March, 1970.

Feurzeig, W., Papert, S., Bloom, M., Grant, R., & Solomon, C. *Programming languages as a conceptual framework for teaching mathematics.* Report No. 1889. Boston: Bolt, Beranek & Newman, 1969.

Feurzeig, W., & Lukas, G. A programmable robot for teaching. *The International Congress of Cybernetics and Systems*, Oxford, England, 1972.

Feurzeig, W., & Lukas, G. Logo: A programming language for teaching mathematics. *Educational Technology*, March, 1972.

Feurzeig, W., Lukas, G., Faflick, P., Grant, R., Lukas, J., Morgan, C., Weiner, W., & Wexelblat, P. *Programming languages as a conceptual framework for teaching mathematics.* Report No. 2165. Final Report, NSF-C-615, Vols. 1-3. Boston: Bolt, Beranek & Newman, 1971.

Fischer, G. *Material and ideas to teach an introductory programming course using Logo.* Irvine, Calif.: Department of Information and Computer Science, U. C. Irvine, 1973.

Fletcher, J. Computer science seminars on pedagogical techniques and methods for evaluation. *Seventh Annual Systems Symposium*, Princeton, March 1973.

Folk, M., Statz, J., & Seidman, R. *Syracuse university Logo project.* Report No. 3. Final Report, NSF-TIE-GJ32222-3. Syracuse, New York: Syracuse University, 1974.

Forsythe, A., Keenan, T., Organick, E., & Stenberg, W. *Computer science: A first course.* New York: Wiley, 1969.

Fromkin, V. Slips of the tongue. *Scientific American*, December, 1973.

Gardner, R., & Gardner, B. Early signs of language in child and chimpanzee. *Science*, 187, 752-753, 28 February, 1975.

Ginsberg, S. *The mathematical theory of context-free languages.* New York: McGraw-Hill, 1966.

Givens, W. Implications of the digital computer for education in the mathematical sciences. *Communications of the ACM*, September, 1966.

Goldberg, A. *Computer-assisted instruction: The application of theorem proving to adaptive response analysis.* Technical Report No. 203. Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Goldberg, A., Levine, D., & Weyer, S. Three sample instructional programs from Stanford University. *Computers in the instructional process: Report of an international school.* Ann Arbor, Mich.: Extend Publications, 1974.

Goldstein, I. *Understanding simple picture programs.* Technical Report No. 294. Cambridge, Mass.: Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1974.

Gries, D. *Compiler construction for digital computers.* New York: Wiley, 1971.

Harlan, J. Our vanishing genetic resources. *Science*, 188, 618-621, 9 May, 1975.

Hoare, C. Proof of a program Find. *Communications of the ACM*, January, 1971.

Holton, G. On the role of themata in scientific thought. *Science*, 188, 328-334, 25 April, 1975.

Jackson, P. *Life in classrooms.* New York: Holt, Rinehart and Winston Inc., 1968.

Jaynes, E. Confidence intervals versus Bayesian intervals. *The International Symposium on Foundations of Probability and Statistics and Statistical Theories of Science*, University of Western Ontario, May, 1973.

Kay, A. *Personal dynamic media.* Xerox Learning Research Group, Xerox Palo Alto Research Center, Palo Alto, CA., June, 1975.

Kay, A. A personal computer for children of all ages. *Proceedings of the ACM National Conference*, Boston, 1972.

Kay, A. A dynamic medium for creative thought. *Meeting of The National Council of Teachers of English*, Minneapolis, 1972.

Kimball, R. *Self-optimizing computer-assisted tutoring: Theory and practice.* Technical Report No. 206. Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1973.

Knuth, D. *MIX.* Reading, Mass.: Addison-Wesley Series in Computer Science and Information Processing, 1970.

Knuth, D. *Fundamental Algorithms.* Reading, Mass.: Addison-Wesley, 1968.

Koestler, A. *The roots of coincidence.* New York: Vintage Books/Random House, 1973.

Kolata, G. Communicating mathematics: Is it possible? *Science*, 187, 732, 28 February, 1975.

Kolers, P. Experiments in reading. *Scientific American*, July 1972.

Kruskal, W. Letter to the editor, *Science*, 188, 10, 4 April, 1975.

Kruskal, J. Multidimensional scaling by optimizing goodness of fit to a nonmetric hypothesis. *Psychometrika*, 29, 1-27, 1964.

Ledgard, H. Ten mini-languages: A study of topical issues in programming languages. *Computing Surveys*, 3, 115-146, September, 1971.

Levison, M., Ward, G., & Webb, J. *The settlement of Polynesia. A computer simulation.* Minneapolis: University of Minnesota Press, 1973.

Lorton, P., & Muscat, E. Computer utilization at the secondary school level: A model for computer assisted career education. In G. Goos & J. Hartmanis (Eds.), *Lecture notes in computer science.* 17, Berlin: Springer-Verlag, 1975.

Lorton, P., & Slimick, J. Computer-based instruction in computer programming. *Fall Joint Computer Conference*, Las Vegas, 1969.

Manis, V. *A machine independent implementation of Logo.* Unpublished doctoral dissertation, University of British Columbia, 1973.

Manna, Z. *Introduction to the mathematical theory of computation.* New York: McGraw-Hill, 1972.

Mauchly, J. Mauchly on the trials of the Eniac. *IEEE Spectrum*, April, 1975.

Maurer, W. *Programming.* San Francisco: Holden-Day, 1968.

McLaughlin, M. *Evaluation and Reform: The elementary and secondary education act of 1965, Title I.* Report No. R-1292-RC. Santa Monica, Calif.: The Rand Corporation, 1974.

Merton, R. Thematic analysis in science: Notes on Holton's concept. *Science*, 188, 335-337, 25 April, 1975.

Milner, S. The effects of computer programming on performance in mathematics. *Annual Meeting of the AERA*, New Orleans, February, 1973.

Minsky, M. (Ed.) *Semantic information-processing.* Cambridge, Mass.: MIT Press, 1968.

Minsky, M. *Computation: Finite and infinite machines.* Englewood Cliffs, N.J.: Prentice-Hall, 1967.

Morris Jr., J. Another recursion induction principle. *Communications of the ACM*, May, 1971.

Nedelsky, L. Evaluation of essays by objective tests. *The Journal of General Education*, 7, 209-220, April, 1953.

Newell, A., & Simon, H. *Human problem solving.* Englewood Cliffs, N. J.: Prentice-Hall, 1972.

Nievergelt, J. The automation of introductory computer science courses. *The International Computing Symposium*, Davos, Switzerland, September, 1973.

Nilson, N. *Problem solving methods in artificial intelligence.* New York: McGraw-Hill, 1971.

Oettinger, A., & Marks, S. *Run, computer run: The mythology of educational innovation.* Boston: Harvard Press, 1969.

Papert, S. Teaching children thinking. *IFIP Conference on Computer Education*, Amsterdam, August, 1970.

Piaget, J. *Genetic epistemology.* New York: Columbia University Press, 1970.

Pirsig, R. *Zen and the art of motorcycle maintenance.* New York: Morrow & Co., 1974.

Polya, G. *How to solve it.* Princeton, N.J.: Princeton University Press, 1957.

Puri, M. (Ed.) *Nonparametric techniques in statistical inference.* London: Cambridge University Press, 1970.

Reisfeld, R., & Kahan, B. Markers of biological individuality. *Scientific American*, June, 1972.

Reynolds, J. Gedanken: A simple typeless language based on the principle of completeness and the reference concept. *Communications of the ACM*, May, 1970.

Roman, R. *Logo: A student manual.* Pittsburgh: Learning Research and Development Center, University of Pittsburgh, 1972.

Rozeboom, W. The fallacy of the null hypothesis significance test. In D. Morrison & R. Henkel (Eds.), *The significance test controversy.* Chicago: Aldine Publishing Co., 1970.

Rubinstein, R. *Computers and a liberal education: Using Logo at the undergraduate level.* Irvine, Calif.: Department of Information and Computer Science, U. C. Irvine, 1974.

Salomaa, A. *Theory of automata.* London: Pergamon Press, 1969.

Scribner, S., & Cole, M. Cognitive consequences of formal and informal education. *Science*, 182, 553-559, 9 November, 1973.

Serin, J. Letter to the editor. *Science*, 189, 86-88, 11 July, 1975.

Smallwood, R. *A decision structure for teaching machines.* Boston: MIT Press, 1962.

Steel, T. (Ed.) *Formal-language description-languages.* Amsterdam: North-Holland, 1966.

Steen, L. Foundations of mathematics: Unsolvable problems. *Science*, 189, 209-210, 18 July, 1975.

Stent, G. Limits to the scientific understanding of man. *Science*, 187, 1052-1057, 21 March, 1975.

Stent, G. Letter to the editor, *Science*, 189, 504, 15 August, 1975.

Swinehart, D., & Sproull, R. *SAIL.* Sailon No. 57.2, Stanford, Calif.: Stanford Artificial Intelligence Laboratory, 1971.

Toomre, A., & Toomre, J. Violent tides between galaxies. *Scientific American*, December, 1973.

Tukey, J., & Wilk, M. Data analysis and statistics: Techniques and Approaches. *The Symposium on Information Processing in Sight Sensory Systems.* California Institute of Technology, November, 1965.

Vacroux, A. Microcomputers. *Scientific American*, May, 1975.

Weyer, S., & Cannara, A. *Children learning computer programming: Experiments with languages, curricula and programmable devices.* Technical Report No. 250. Stanford, Calif.: Institute for Mathematical Studies in the Social Sciences, Stanford University, 1975. ERIC #ED-111347.

Weyer, S., & Cannara, A. Programming projects for children: Graphics and speech synthesis. *The Second Annual Computer Science Conference*, Detroit, February, 1974.

Winograd, T. When will computers understand people? *Psychology Today*, May, 1974.

Winograd, T. *Procedures as a representation of data in a computer program for understanding natural language.* Project MAC TR-84. Cambridge, Mass.: Massachusetts Institute of Technology, 1971.

Wirth, N. Euler: A generalization of Algol, and its formal definition. *Communications of the ACM*, February, 1966.

Wittrock, M. (Ed.) *Changing education: Alternatives from educational research.* Englewood Cliffs, N.J.: Prentice-Hall, 1973.

Worthen, B., & Sanders, J. *Educational evaluation: Theory and practice.* Worthington, Ohio: Charles Jones Publishing Co., 1973.

Appendix 1 Spm

This appendix documents the syntax and semantics of a language (Spm) designed by the author but never used in any experiments. It simulates a string-processing machine in which one operation, assignment ('ISNOW'), plays the central role. First, a comparison of Logo/Spm phrasing:

Litera	"DONALD"	[DONALD]
Name/value linking	MAKE "DONALD" "DUCK"	@[DONALD] ISNOW [DUCK] or DONALD ISNOW [DUCK]
Name evaluation	THING OF "DONALD" or just :DONALD:	@[DONALD] or DONALD
Indirect naming	MAKE :DONALD: "FOWL"	@@[DONALD] ISNOW [FOWL] or @DONALD ISNOW [FOWL]
String appending	WORD OF "ABC" AND "D"	STACK ISNOW [ABC]; STACK ISNOW [D]; NEXT ISNOW APPEND;
String definitions (substitutions)	ABBREVIATE "WORD" AS "JOIN"	[PUSH] ISFOR [STACK ISNOW]; [DO] ISFOR [NEXT ISNOW]; [TYPE] ISFOR [TTY ISNOW];
Input/output	PRINT JOIN "ABC" "D"	PUSH [ABC]; PUSH [D]; DO APPEND; TYPE STACK;
L .belling	line numbers as below	LAB ISNOW NEXT; DO LAB;
Storage release	automatic	FORGET LAB;
Operation definition (dialogs)	←TO DD :W: :C: @10 PRINT JOIN :W: :C: @END	DD ISNOW [DO APPEND; TYPE STACK; @[] ISNOW NEXT];
Execution call	←DD "ABC" "D" ABCD	PUSH [ABC]; PUSH [D]; DO DD;ABCD
Recursion	←TO RECURSE @10 P "RECURSE" @20 RECURSE @END ←RECURSE RECURSE :	RECURSE ISNOW [TYPE [RECURSE]; DO RECURSE;]; DO RECURSE;RECURSE...

Spm Syntax. The meta-symbols ← and | mean, respectively, "rewrite as" and "or". The paired meta-symbols < > ( ) and { } mean, respectively, "a non-terminal", "one of" and "optional". Spaces may be ignored. Note that ( ) < > and ← appear both as terminal and meta-symbols:

```
<program>          ← {<blank>} {<statement> {<blank>}} ; {<program>}

<blank>            ← <non-printing teletype motion character> {<blank>}

<statement>        ← <comment> | <assignment> | <substitution> | <forget> |
                     <test> {<blank>} : {<blank>} <statement>

<comment>          ← <literal> {{<blank>} <comment>}

<assignment>       ← <destination> ISNOW (<source> | {<blank>} <literal>)

<substitution>     ← (<destination> | <literal> {<blank>}) ISFOR
                     (<source> | {<blank>} <literal>)

<forget>           ← FORGET <source>

<test>             ← (IFEMPTY | IFNOTEMPTY) (<source> | <literal>)

<literal>          ← [ <balanced string> ]

<balanced string>  ← {<string>} {<balanced string>} {<literal>} {<string>}

<destination>      ← (<name> | <indirect name>) <blank> |
                     <literal name> {<blank>}

<source>           ← <blank> <name> |
                     {<blank>} (<literal name> | <indirect name>)

<literal name>     ← @ {<blank>} (<literal> | <literal name>)

<indirect name>    ← @ {<blank>} (<name> | <indirect name>)

<string>           ← <blank> | (<name> | : | ; | @ ) {<string>}

<name>             ← ( A | B | C | D | E | F | G | H | I | J | K | L | M |
                     N | O | P | Q | R | S | T | U | V | W | X | Y | Z |
                     0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ! | " | # |
                     $ | % | & | ' | ( | ) | * | + | , | - | . | / | < |
                     = | > | ? |       | ↑ | ← ) {<name>}
```

Semantics. The Spm machine consists of (a) a processor, which interprets strings in the Spm language defined above; (b) an arbitrary number of memory cells, each capable of storing a balanced string of arbitrary length; and (c) two pushdown stores called 'STACK' and 'NEXT', each consisting of an ordered set of memory cells of which only that cell bearing the name of the set is immediately accessible. New memory cells are created as needed to satisfy assignments. Any existing accessible cells, except 'STACK' and 'NEXT', can be released. The Spm machine also maintains an inaccessible and variable stock of cells for satisfying 'ISFOR' statements and 'STACK' and 'NEXT' manipulations.

Certain strings have special meaning to the Spm machine as names, primitive operations. Names which Spm automatically associates with specific memory cells are: 'TTY', 'STACK' and 'NEXT'. Other names are defined by the execution of assignments. All accessible cells must necessarily have distinct names. Spm operations are predefined names which cause specific activities of the machine when it encounters them during the execution of statements. 'ISNOW', 'ISFOR', 'FORGET', 'IFEMPTY', 'IFNOTEMPTY', '@', ';', ':', '[' and ']' have such effect. The two pushdown stores, whose accessible cells are named 'STACK' and 'NEXT', have special properties: (a) if either 'STACK' or 'NEXT' appears as the destination in an assignment, the machine attaches a new cell to the accessible end of the appropriate ordered set of cells. The new cell is loaded with the value of the source and the name 'STACK', or 'NEXT' as appropriate, is associated with this new cell rather than with the previously accessible cell; (b) if either 'STACK' or 'NEXT' occurs as a source in an assignment, a substitution or a <forget>, the Spm machine uses the accessible cells's content, and releases the cell. The name 'STACK' or 'NEXT', as appropriate, is then associated with the next cell in the corresponding ordered set of cells. The same action results when either 'STACK' or 'NEXT' appears as a destination in a substitution statement; (c) no change in the structure of 'STACK' or 'NEXT' is made if either appears as a source in a test.

Spm Primitives

Symbols	Operations
NEXT	ISNOW
STACK	ISFOR
TTY	FORGET
	IFEMPTY
	IFNOTEMPTY
	APPEND
	HEAD
	TAIL
	AFTER
	@
	;
	:
	[
	]

'NEXT" always contains the string to be executed next by the Spm machine. The machine obtains one statement after another from this string by scanning the value of 'NEXT" from left to right until a ';', not part of a literal, is encountered. The scanning process removes all characters up through the ';' from 'NEXT', shortening its content as execution procedes. When the last statement in 'NEXT' has been executed, the current cell is released and replaced by that directly beneath it. Should 'NEXT' ever be exhausted of cells, the Spm machine will automatically attempt to fill 'NEXT" with characters from the teletypewriter ("TTY'). If a statement cannot be executed, the machine prints a message and again goes to the teletypewriter for input. Note that this is analogous to execution of the statement 'NEXT ISNOW TTY'. 'STACK' is the accessible cell in the general pushdown store and may have as value any string. 'TTY' is the user's terminal. Assignment to it causes the assigned value to be printed. Assignment from it to a destination obtains characters from the typist. Its value is not maintained by Spm, so characters disappear on the way in or out as typing proceeds at the terminal. Its value is '[]' when input or output has been completed.

'ISNOW' is the means for changing the content of the Spm machine's

memory. When a name is used for the first time in an assignment, the machine obtains a new cell in which to store the assigned value and associates the name with this cell. 'ISFOR' is a simple symbol/string substitution mechanism. After it is executed, the Spm machine will automatically substitute, for any occurrences of the value on the left of the 'ISFOR' in the text of any statement scanned from 'NEXT', the value on the right. This amounts to a simple transformation of the Spm language to suit the user. Recursive substitutions are not allowed. 'FORGET" is the means for releasing names and their associated memory cells from the Spm machine's memory.

'IFNOTEMPTY' and 'IFEMPTY' are tests which, if the value tested is not '[]' or is '[]', respectively, will execute the subsequent statement. Otherwise, the statement is skipped.

'APPEND' joins a character to the end of a string. The character is assumed to be in the top cell of 'STACK', with the string immediately beneath. It returns the resultant string as the value of 'STACK'. If a string is used for the character, only its first character will be appended. 'HEAD' accepts a string in 'STACK' and returns the first character of that string in 'STACK'. 'TAIL' is like 'HEAD', but returns all characters in the string after the first is removed. For 'AFTER', 'STACK' and the cell beneath it each contain a character. If the character in 'STACK' occurs before the other character in the lexicographical ordering defined for the characters of the Spm alphabet, the top character is removed from 'STACK'. Otherwise, both 'STACK' cells are removed and the value of '[]' given to 'STACK'. If strings are supplied as values, only the first character of each will enter into the comparison.

'@' indicates that the value of the string which follows should be interpreted as a name. Note '@[ABC]' and 'ABC' are equivalent. ';' terminates an Spm statement. ':' indicates the beginning of a statement in a test. '[' and

']' respectively denote the start and end of a literal. Note that '[' and ']' must occur in pairs according to the syntax. There is no legal way to obtain either bracket singly in a piece of executable text. They may be obtained from '[]' however, with the 'HEAD' and 'TAIL' functions. Some details of Spm phrasing follow:

String constant	[DONALD] (literal)
Empty string	[]
Assignment	@[DONALD] ISNOW [DUCK] or DONALD ISNOW [DUCK]
Name evaluations	@[DONALD] or just DONALD both have the value DUCK
Recursive naming (unlimited indirect addressing)	@@[DONALD] ISNOW [FOWL] or @DONALD ISNOW [FOWL]
Using 'STACK' (produces):	STACK ISNOW [ABC]; STACK ISNOW [D]; +-----+ | D |<-- STACK +-----+ | ABC | +-----+
Using 'NEXT' (program control, produces):	NEXT ISNOW APPEND; TTY ISNOW STACK; +---------------------+ +-------+ | TTY ISNOW STACK |<-- NEXT | ABCD |<-- STACK +---------------------+ +-------+
Substitutions ISNOW];	[PUSH] ISFOR [STACK ISNOW]; [DO] ISFOR [NEXT [TYPE] ISFOR [TTY ISNOW];
Input/output (types out "ABCD")	PUSH[ABC]; PUSH[D]; DO APPEND; TYPE STACK;ABCD
Storage release	FORGET DONALD;
Operation defining	DD ISNOW [DO APPEND; TYPE STACK;];
Execution (types out "ABCD")	PUSH [ABC]; PUSH [D]; DO DD;ABCD
Recursion	RECURSE ISNOW [TYPE [ABCD]; DO RECURSE;]; DO RECURSE;ABCDABCDABCDABCD...
Stack release	@[] ISNOW STACK;
Premature return	@[] ISNOW NEXT;

Appendix 2 Aptitude-Testing Details

An Example of Commercial Test Evaluation

The example derives from remarks in the published manual for one of the programming tests examined. The validity of that test was assessed by three studies: (1) correlation of test scores and grades of three groups of programming trainees, (2) correlation of test scores and overall performance ratings by supervisors of programmers, and (3) a study like that of (2) in which grades on a training course were also available. Studies (1) and (3) both assumed, without discussion, that the testing done during training was itself a valid measure of programming ability. Studies (2) and (3) both assumed that ratings by superiors were similarly valid. Study (1) indicated that, of fifteen relevant correlations between subtest scores and trainee groups, eight were of statistical (normal theory) significance. And only one subtest was significantly correlated with trainee performance over all groups, in spite of the fact that the overall test/training correlation for each group was significant. Interestingly, the most variable subtests were those which relied heavily on time and repetition. In Study (2), three of five subtest correlations and the overall correlation were significant but small; and the two remaining subtests were those which exhibited variable or minimal correlation with performance in study (1). Unfortunately, the ratings used as the validating measure in (2) were not confined to programming ability and included such things as attitudes. Therefore, study (2) is invalid. Study (3) found three subtests significantly correlated with training course grades, but one of the three had not been significantly correlated with grades for any group in study (1). Furthermore, the ratings used in the other half of study (3) were virtually uncorrelated with subtest results. The brochure went on to state that these ratings and job tenure were correlated more strongly than anything else in both halves of the study--the suggestion being that low correlations must be expected wh

evaluations place high value on relatively invalid properties (i.e., tenure). An alternative observation can be made which applies to any correlational procedure: the sample variance of a measured property may be so low that apparent but spurious correlations with another measure arise. In study (3), the test scores could have had low variability for good reason: the testees could have been of very nearly the same competence. In any event, none of the studies provided a clear validation of this particular test for programming aptitude.

1974 Test Questions

The pre- and post-tests given to students in the second experiment are presented here, beginning on pages 127 and 131 respectively (some of the questions are specifically referred to in the text). All students in that experiment worked the pretest, but only a few, who finished both the Logo and Simper curricula, worked the posttest. The questions in these tests were drawn from the same set used to construct the 1973 experiment's pretests and so reflect their content as well.